STATPACK Documentation

Release 2.4

Pascal Terray (IRD)

Oct 02, 2025

CONTENTS

1 Introduction **1**
- 1.1 Presentation . . . 1
- 1.2 Language . . . 2
- 1.3 Parallelism and BLAS . . . 3

2 Installation **7**
- 2.1 Basic installation . . . 7
- 2.2 OpenMP compilation . . . 15
- 2.3 Preprocessor cpp macros . . . 16

3 STATPACK overview **19**
- 3.1 sources directory . . . 19
- 3.2 tests directory . . . 23
- 3.3 examples directory . . . 24
- 3.4 doc directory . . . 25
- 3.5 interfaces directory . . . 25
- 3.6 makeincs directory . . . 25
- 3.7 myprograms directory . . . 25

4 Using the STATPACK library **27**
- 4.1 Example program . . . 27
- 4.2 Compiling and linking . . . 30
- 4.3 Shared libraries . . . 30
- 4.4 Parallel execution . . . 31
- 4.5 Using long integers . . . 33

5 STATPACK reference manual **35**
- 5.1 Introduction . . . 35
- 5.2 MODULE The_Kinds . . . 36
- 5.3 MODULE Select_Parameters . . . 37
- 5.4 MODULE Derived_Types . . . 39
- 5.5 MODULE Reals_Constants . . . 39
- 5.6 MODULE Logical_Constants . . . 42
- 5.7 MODULE Char_Constants . . . 43
- 5.8 MODULE Num_Constants . . . 44
- 5.9 MODULE Sort_Procedures . . . 48
- 5.10 MODULE Print_Procedures . . . 50
- 5.11 MODULE String_Procedures . . . 53
- 5.12 MODULE Time_Procedures . . . 57
- 5.13 MODULE Utilities . . . 61

5.14 MODULE Utilities_With_Pnter . . . 72
5.15 MODULE Random . . . 73
5.16 MODULE Giv_Procedures . . . 90
5.17 MODULE Hous_Procedures . . . 97
5.18 MODULE QR_Procedures . . . 101
5.19 MODULE Eig_Procedures . . . 110
5.20 MODULE SVD_Procedures . . . 132
5.21 MODULE LLSQ_Procedures . . . 183
5.22 MODULE Lin_Procedures . . . 189
5.23 MODULE Prob_Procedures . . . 198
5.24 MODULE Stat_Procedures . . . 212
5.25 MODULE Mul_Stat_Procedures . . . 222
5.26 MODULE FFT_Procedures . . . 236
5.27 MODULE Time_Series_Procedures . . . 241
5.28 MODULE BLAS_interfaces . . . 265
5.29 MODULE Lapack_interfaces . . . 268
5.30 MODULE Statpack . . . 274

6 STATPACK modules manuals **277**
6.1 Module_BLAS_Interfaces . . . 277
6.2 Module_Char_Constants . . . 277
6.3 Module_Derived_Types . . . 278
6.4 Module_Eig_Procedures . . . 279
6.5 Module_FFT_Procedures . . . 393
6.6 Module_Giv_Procedures . . . 409
6.7 Module_Hous_Procedures . . . 431
6.8 Module_LLSQ_Procedures . . . 445
6.9 Module_Lapack_Interfaces . . . 490
6.10 Module_Lin_Procedures . . . 491
6.11 Module_Logical_Constants . . . 539
6.12 Module_Mul_Stat_Procedures . . . 540
6.13 Module_Num_Constants . . . 633
6.14 Module_Print_Procedures . . . 641
6.15 Module_Prob_Procedures . . . 652
6.16 Module_QR_Procedures . . . 735
6.17 Module_Random . . . 763
6.18 Module_Reals_Constants . . . 836
6.19 Module_SVD_Procedures . . . 837
6.20 Module_Select_Parameters . . . 1067
6.21 Module_Sort_Procedures . . . 1068
6.22 Module_Stat_Procedures . . . 1075
6.23 Module_Statpack . . . 1144
6.24 Module_String_Procedures . . . 1145
6.25 Module_The_Kinds . . . 1159
6.26 Module_Time_Procedures . . . 1160
6.27 Module_Time_Series_Procedures . . . 1168
6.28 Module_Utilities . . . 1283
6.29 Module_Utilities_With_Pnter . . . 1344

Bibliography **1351**

Index **1361**

CHAPTER

ONE

INTRODUCTION

1.1 Presentation

STATPACK is a Fortran 95/2003 multi-threaded library for solving the most commonly occurring mathematical and statistical problems in the processing of climate model outputs and datasets, and, more generally, in the analysis of huge datasets found in a large variety of scientific fields. It is a freely-available software, and started from version 2, STATPACK is released under the GNU LGPL license. Details about the license can be found at LGPL License.

All the information related to STATPACK can be found at the following web site STATPACK. The distribution tar file of the software is available for download at this site. Other information is provided there, such as installation instructions and contact information. Instructions for installing the software can also be found below, in the chapter *Installation*.

The distribution tar file of STATPACK contains the Fortran 95/2003 sources of the library and the associated test and example programs. It also contains an ensemble of portable makefiles, which allows fast and automatic compilation of the STATPACK library on most UNIX/Linux systems, AIX and Mac OSX. This ensemble of portable makefiles also defines a friendly and useful environment for compiling (and executing) Fortran 95/2003 programs on a computer.

The routines available in STATPACK currently include (in version 2.4):

- Numerical linear algebra subroutines for statistical computations (LU, Cholesky, QR, QL and QLP decompositions, linear solvers, least squares solvers, full and partial eigenvalue and singular value decompositions, generalized inverses of full or symmetric matrices, determinant of a square matrix, . . .)
- Randomized and deterministic (full or partial) QR decomposition with Column Pivoting (QRCP), Complete Orthogonal Decomposition (COD) or QB decomposition of a matrix and associated linear least squares solvers
- State-of-the-art approximate partial eigenvalue and singular value decompositions based on randomized power subspace and block Krylov iterations, or a preliminary QRCP or QLP factorization
- A large set of very fast randomized or deterministic routines for solving accurately the fixed-rank and fixed-precision problems
- Randomized and deterministic routines for computing the column Interpolative (ID), the two-sided Interpolative (tsID) and CUR decompositions of a matrix
- Probability functions and their inverses
- Out of core statistical univariate and multivariate functions and subroutines
- Random number or matrix generation and Monte Carlo procedures
- Fast Fourier Transforms for both real and complex data of general length
- Time series analysis functions and subroutines
- Utilities for printing and sorting matrices and vectors
- Utilities for manipulating strings, dates and times

- ...

The emphasis of the software is on fast and robust methods appropriate for problems in which the associated matrices are large and dense, for example, those arising in the statistical analysis of very high resolution climate model outputs for which standard commercial or free softwares currently used in the climate scientific community (e.g. *MATLAB*, *IDL*, *Grads*, *Ferret*, *Ncl*, *Python*, ...) may have difficulties.

Note also that prior to building STATPACK library, none other packages are required or must be installed. In other words, STATPACK is a fully portable software and the efficiency of STATPACK on a specific machine does not depend directly on the implementation of other pre-installed softwares such as the BLAS (e.g., Basic Linear Algebra Subprograms; [blas]), but only on the quality/performance of the Fortran 95/2003 compiler (in particular of the performance of the built-in intrinsic procedures like the Fortran 90 **matmul()** and **dot_product()** functions) and its OpenMP [openmp] parallelism features. Optionally, however, the STATPACK library can also greatly benefit from an optimized/multi-threaded BLAS library [blas] for enhanced performance at the user option. Many open source or vendor versions of the BLAS library are currently available depending on your system and can be used in conjunction with STATPACK to enhance its performance [openblas] [atlas] [blis] [mkl] [aocl] [arm] [accelerate] ...

With OpenMP and BLAS supports activated, the current version of STATPACK delivers maximum efficiency and can compete with state-of-the-art Dense Linear Algebra libraries such as OpenBLAS [openblas], LIBFLAME [libflame], Intel Math Kernel Library [mkl], AMD Optimizing CPU Libraries [aocl], Arm Performance Libraries [arm] or Apple Accelerate framework [accelerate] for many problems, especially those involving the computations of the full or partial Singular Value Decomposition (SVD) of a real matrix or those related to the full or partial EigenValue Decomposition (EVD) of a real symmetric matrix. See the section *Parallelism and BLAS* for more details on how activating OpenMP and using an optimized BLAS library.

STATPACK has been built successfully on a variety of UNIX systems (including Mac OSX and AIX) and with many different Fortran 95/2003 compilers. It is believed that STATPACK is a portable software. Currently, STATPACK does not use Graphics Processing Units (GPUs), but this possibility may be included in future releases of STATPACK.

Furthermore, with BLAS support activated and use of the NVBLAS library from NVIDIA [nvblas] which is a multi-GPUs accelerated library that implements BLAS level-3 kernels and can be used as drop-in BLAS level-3 replacement on systems with both GPUs and CPUs, the STATPACK library can also already benefit from GPUs present in the system. More precisely, depending on the charateristics of the BLAS level-3 calls in the STATPACK library, NVBLAS will redirect the calls to the GPUs present in the system or to the CPU. See the documentation of the NVBLAS library for more details on its use [nvblas].

Finally, for most users in the climate community, best flexibility and usability are simply achieved with the use of the NCSTAT software [ncstat] in addition to the STATPACK library. The NCSTAT software is a collection of many UNIX stand-alone operators for statistical processing and analysis of huge climate model outputs and datasets stored in the NetCDF format [netcdf]. These stand-alone operators are also written in Fortran 95/2003 using the NetCDF Fortran 90 interface [netcdf-f90] of the NetCDF library [netcdf] for input/output data transfer and the STATPACK software for numerical and parallel computations. More information about NCSTAT can be found at the following web site NCSTAT.

1.2 Language

The STATPACK library consists of numerical algorithms written in pure and portable Fortran 95/2003 language constructs without any obsolescent Fortran77 features [Fortran]. STATPACK uses all the new features of the Fortran 95/2003 standard, including:

- Fortran 90 array data types
- Symbolic names for the parameterization of the kind parameters for real, complex, integer and logical data
- Fortran 90 modules
- Fortran 90 explicit interfaces

- Overloading of procedures and functions for ease of use (e.g., generic routines)
- Allocatable and automatic arrays (dynamical storage)
- Optional arguments to functions and subroutines
- Assumed-shape arrays for arguments passing
- ...

At the user option, new Fortran 2003 constructs and intrinsic modules, like the *IEEE_EXCEPTIONS*, *IEEE_ARITHMETIC* and *IEEE_FEATURES* modules [Fortran], can also be used in STATPACK. See the section *Preprocessor cpp macros* for more details.

Thus, to use this product you should be familiar with the Fortran 95/2003 language and you must have access to a Fortran 95/2003 compiler to build the STATPACK library. For more information on Fortran 95/2003, see [Fortran] or consult one of the many tutorials available on the Web, for example Fortran tutorial.

While many current standard Fortran packages, like LAPACK or SCALAPACK (e.g., two famous libraries of Fortran routines for solving problems in numerical linear algebra on shared-memory and distributed-memory architectures, respectively), include different versions of the subroutines and functions for different Fortran data types (e.g., real or complex single- and double-precision arithmetic), the systematic use of Fortran 95/2003 parameterized data types in STATPACK allows the user to choose exactly the precision of the version of the STATPACK library he wants [Buckley:1994a] [Buckley:1994b]. See the Fortran program ex1_svd_cmp.F90 for an illustration of the use of Fortran 95/2003 parameterized data types in STATPACK.

Currently, it is possible to build single, double and also quadruple precision versions of STATPACK, as well as specific versions requesting precise and portable precision specifications for real and complex computations included in the software [Buckley:1994a]. The choice for the precision specification for a particular version of the library is done when building the library. See the chapter *Installation* for more details.

The interface loading features of the Fortran 95/2003 language are also heavily used in STATPACK, an useful feature, which is also missing in standard Fortran packages. As an illustration, the generic `solve_lin()` function exported by the module *Lin_Procedures*, which can be used to solve a linear system with one or multiple right hand sides, accepts the following calls:

```
x(:n)     = solve_lin( mat(:n,:n) , b(:n)     , tol=tol )
x(:n,:m) = solve_lin( mat(:n,:n) , b(:n,:m) , tol=tol )
```

The advantages of the overloading are obvious since the same interface is used for one or several right hand sides, or for different precisions.

1.3 Parallelism and BLAS

STATPACK is a parallel, multi-threaded software based on the OpenMP standard. Therefore, it will run on multi-core or, more generally, shared-memory multi-processor computers. It is also possible to build sequential versions of STATPACK (e.g., if OpenMP compilation is disabled or if an OpenMP-enabled Fortran compiler is not available), even if it is not at all recommended for efficiency reasons. STATPACK does not run on distributed memory (e.g., clusters) parallel computers.

In both the LAPACK [lapack] and ScaLAPACK [scalapack] libraries, the exploitation of parallelism comes from the availability of a parallel BLAS implementation. In the LAPACK case, a number of BLAS libraries can be used to take advantage of multiple processing units on shared-memory systems; for example, the freely distributed ATLAS [atlas], GotoBLAS [gotoblas], OpenBLAS [openblas], BLIS [blis] libraries or other vendor BLAS like Intel Math Kernel Library [mkl], Arm Performance Libraries [arm], AMD Optimizing CPU Libraries [aocl] or Apple Accelerate framework [accelerate] are popular choices. In the ScaLAPACK case, parallelism is exploited by PBLAS [pblas], which is a parallel BLAS implementation that uses the Message Passing Interface (MPI; [mpi]) for communications on a distributed memory system.

In both cases, parallelism is enclosed inside the BLAS routines for most implementations of LAPACK or ScaLAPACK. In a typical multi-core implementation this means that each BLAS routine contains at least one parallel section and for each call to BLAS, a whole set of threads is started and stopped at least with each BLAS call. This thread management overhead is relatively small for level-3 BLAS routines [blas3], but it could be very significant for level-1 and -2 BLAS operations [blas1] [blas2] due to the low computational intensity in these level-1 and -2 BLAS kernels.

On the other hand, STATPACK does not rely directly on the BLAS to obtain high performance in its numerical Dense Linear Algebra subroutines, since STATPACK uses parameterized data types and allows the user to build quadruple-precision version of the library (remember that standard BLAS, LAPACK and ScaLAPACK libraries exist only in single- and double-precision). Instead, relatively good performance is obtained in STATPACK by reformulating old algorithms or developing new algorithms in a way that their implementations can be easily mapped on recent multi-core systems and take advantage of shared-memory parallelization at a level well-above to the BLAS level. To the best of our knowledge, only the vendor LAPACK versions included in the Intel Math Kernel Library [mkl], Numerical Algorithm Group Library [nag] and Arm Performance Libraries [arm] have improved multi-core and OpenMP implementations of some of its Dense Linear Algebra drivers well above the BLAS level, especially those related to the QR factorization, EigenValue Decomposition (EVD) or Singular Value Decomposition (SVD).

In this spirit, most of the computer intensive methods offered by the STATPACK library are parallelized with the OpenMP Application Program Interface (OpenMP API; [openmp]), which is one of the most common techniques for shared-memory parallelization available with modern Fortran 95/2003/2008 compilers. The OpenMP API is a collection of compiler directives, library routines, and environment variables that can be used to specify shared-memory parallelism in C, C++ and Fortran programs. More information about OpenMP can be found at the following web site OpenMP or in the more friendly tutorial available at OpenMP tutorial. The best place to view OpenMP support by a large range of Fortran compilers is OpenMP compilers. Support for at least OpenMP 3 standard is requested for activation of OpenMP parallelism in version 2.4 of STATPACK and most Fortran compilers are currently supporting the OpenMP 3 standard.

As explained above, the STATPACK library has been designed to take advantages of OpenMP shared-memory parallelism at a high level (e.g., well above the BLAS level) in the algorithms offered in the software. In most cases, this means that each STATPACK routine contains only one or two "global" OpenMP parallel regions and the critical parts of the algorithm are implemented with OpenMP synchronization and barrier primitives inside each parallel region. The programming challenge is thus to minimize the number of these artificial synchronization points in each OpenMP parallel region/routine. In this way, STATPACK users can benefit of good speedup in their programs on all (shared-memory) parallel computers for all choices of the parameterized Fortran real/complex data types (and not only on Intel compatible platforms on which Intel Math Kernel Library is the standard choice for Dense Linear Algebra computations in single- or double-precision).

Note, however, that the parallel paradigm used in STATPACK is still based on the classical fork-and-join scheduling model and, thus, differs from the more advanced methods used in the ongoing PLASMA project. PLASMA lays out matrices in small square tiles, such that each tile occupies a continuous memory region, and is based on new algorithms working on tiles [plasma] [Abalenkovs_etal:2017] [Dongarra_etal:2019].

Currently, PLASMA also relies on the BLAS and OpenMP for dynamic, task-based, scheduling [YarKhan_etal:2016] [Dongarra_etal:2019], but offers only a collection of routines for solving linear systems of equations and linear least squares problems [Dongarra_etal:2019], which are not sufficient for the goals of STATPACK. Furthermore, at least OpenMP 4.0 is required for compiling recent versions of PLASMA and not all Fortran compilers are currently supporting the full OpenMP 4.0 standard. Additionally, the systematic use of the PLASMA tile format is challenging in many applications despite that PLASMA provides efficient routines to convert from/to the tile format and the conventional column/row order layout of a matrix. This is why STATPACK is not currently using PLASMA for Dense Linear Algebra computations for maximum portability across current Fortran compilers/platforms.

On the other hand, through the use of the OpenMP 3.0 API, it is expected that the STATPACK software will achieve portability to a wide range of platforms with good (parallel) performance and maximum flexibility and usability as STATPACK still uses the conventional column order layout of a matrix, which is the standard in Fortran.

Moreover, optionally for single- and double-precision versions of the library, STATPACK can also benefit from an optimized/multi-threaded BLAS library (e.g., [openblas], [mkl], [blis], ...) and includes generic interfaces for several

drivers in LAPACK [lapack], if optimized versions of these libraries are available on your computer. Note, however, that STATPACK does not share any code with the BLAS and LAPACK packages and it is a completely independent software. An optimized BLAS library will provide very significant enhanced speedup with STATPACK if the quality of your Fortran 95/2003 compiler is not enough to obtain the best performance on your computer. This is typical for many compilers, for example with the GNU *gfortran* compiler, but will also restrict the available precisions of STATPACK since the standard BLAS (and also LAPACK) software is only available in single- and double-precisions. Extended- and quadruple-precision versions of the BLAS and LAPACK libraries are available in QPBLAS and XBLAS packages, but they both use a double-double algorithm in which one quadruple precision number is represented by a combination of two double precision numbers [qpblas] [xblas], which makes their use in conjunction with STATPACK very challenging, if not impossible. On the other hand, IEEE 754- compliant quadruple precision (128 bit) floating point operations have already been implemented in modern Fortran, but many hardwares have not been optimized for those operations yet, excepted for Intel x86 processors in which extended double precision is native and provides improved accuracy at full hardware speed. For other processors, quadruple precision (128 bit) calculation speed is currently extremely slow compared to double precision operations. Despite these caveats, STATPACK can and has been compiled in IEEE 754- compliant quadruple precision (128 bit) without BLAS support and is able to provide OpenMP multi-threaded Dense Linear Algebra routines at this IEEE 754- quadruple precision, which can be useful for many applications in which very accurate results are needed.

Most of the modern Fortran 95/2003 compilers (e.g., *gfortran*, *flang*, *ifort*, *ifx*, *pgfortran*, *nvfortran*, *xlf2003*, *nagfor*, ...) include a simple command line option to the compiler that activates and allows interpretation of all OpenMP directives included in the STATPACK library. If parallel computing is required, it is the responsibility of the user to include the relevant OpenMP command line options to compile the STATPACK library on his (parallel) computer. See the section *OpenMP compilation* below, on how to do this.

Finally, the number of processors used when executing an OpenMP conforming program using the STATPACK library and, more generally, the behaviour of such programs are determined by setting some OpenMP environment variables (e.g., `OMP_NUM_THREADS`, `OMP_MAX_ACTIVE_LEVELS`, `OMP_NESTED`, `OMP_DYNAMIC`, `OMP_STACKSIZE`, ...) just before the execution of the program. See the OpenMP documentation available at OpenMP, OpenMP tutorial or the section *Parallel Execution* for more details and some examples.

CHAPTER

TWO

INSTALLATION

In this chapter, we provide a step by step procedure for the compilation of the STATPACK library. The only requirement is a working Fortran 95/2003 compiler and the availability of the make UNIX tool.

This chapter, and more generally this manual, contains many examples/commands which can be typed at the keyboard. A command entered at the terminal is shown like this:

```
$ command
```

The first character on the line is the terminal prompt, and should not be typed. The dollar sign **$** is used as the standard prompt in this manual, although some systems may use a different character. The examples/commands assume the use of an UNIX-like operating system.

2.1 Basic installation

The basic steps for the installation of STATPACK are described below.

Note that prior to these steps, none other packages are required or must be installed. In other words, STATPACK is a stand-alone software. However, optionally and for efficiency reasons, STATPACK can also benefit of an optimized and multi-threaded BLAS library, if available on your machine, and depending on your choice of the precision used in the library (see below).

It is also possible, optionally, to interface the LAPACK library with STATPACK, again depending on your choice for the precision of the library.

Please follow the following steps for LINUX/Unix systems:

1) Download the latest STATPACK version at STATPACK.

   For example, let us call this package **statpack2.4.tar.gz**.

2) Put the file in your preferred directory such as `$HOME` directory or, for example, `/opt/` directory if you have ROOT privilege.

3) Execute the UNIX command:

   ```
   $ tar -xzvf statpack2.4.tar.gz
   ```

   to decompress the archive. Let us denote <STATPACK *directory*> the package's top directory after decompression. For example, it could be `$HOME/statpack2.4` or `/opt/statpack2.4`.

   This directory, <STATPACK *directory*>, contains the following subdirectories and associated files:

Table 1: Main STATPACK directory

File/subdirectory	Content
makefile	Generic Makefile
make.inc	User specification options for Makefile
LICENSE	STATPACK License file
README	README file
Changelog.txt	Change log file
doc	STATPACK documentation
makeincs	Template `make.inc` files for various compilers/platforms
sources	STATPACK Fortran 90 modules and source code
interfaces	Optional include directory for the `.mod` files generated by the compiler
tests	Testing programs for the STATPACK source code
examples	Example programs for the routines available in STATPACK
myprograms	A directory where, optionally, you can store your own programs using STATPACK

It is not mandatory, but recommended, to set the `STATPACKDIR` Shell environment variable to the path of the STATPACK top directory:

Table 2: Defining the Shell environment variable `STATPACKDIR`

Shell	Command line
csh/tcsh	setenv STATPACKDIR <STATPACK *directory*>
sh/bash	export STATPACKDIR=<STATPACK *directory*>

One of this command can be placed in the appropriate shell startup file in `$HOME` (i.e. `.bashrc` or `.cshrc`

files).

4) In order to proceed to compilation, go to the `$STATPACKDIR` directory:

```
$ cd $STATPACKDIR
```

and edit the `make.inc` file inside this directory and follow the directions to change/specify appropriately:

- the absolute path of the library directory (`DIRLIB`);
- the name of the library (`LIB`);
- the absolute path of the include directory, which will contain the `.mod` files generated by the compiler (`INTERFACES`);
- the name/path of the Fortran 95/2003 compiler (`FORTRAN`);
- the compiler options (`OPTS`, `NOOPTFLAGS`, `OPTFLAGS` and `DRVFLAGS`);
- the archiver and its options to use when building an archive (e.g., a static library; `ARCH` and `ARCHFLAGS`);
- the loader options for your BLAS and LAPACK libraries if you want to use these libraries (`LBLAS` and `LLAPACK`);
- the linker and the flag(s) to use when building a shared library (`LIBTOOL` and `LIBTOOLFLAGS`);
- the loader options for executing the examples and testing programs (`LOADFLAGS`).

Alternatively, you can look at the template `make.inc` examples in the `$STATPACKDIR/makeincs` subdirectory and if one of them matches your compiler/platform, use this file as a template `make.inc` to build your own `make.inc`.

This can be done:

- manually, by overwriting the `make.inc` file in `$STATPACKDIR` by your choice in `$STATPACKDIR/makeincs`;
- by executing the **`make`** command:

  ```
  $ make
  ```

  in the `$STATPACKDIR` directory, selecting the name for your architecture/compiler in the list printed on the screen and, then, executing the **`make`** command:

  ```
  $ make <arch>
  ```

  in the `$STATPACKDIR` directory, where **<arch>** is the selected name for your architecture/compiler. These steps will also overwrite the `make.inc` file in `$STATPACKDIR` by your choice in `$STATPACKDIR/makeincs`.

After these steps, you still need to customize this new `make.inc` file, at least to provide:

- the absolute path of the library directory (`DIRLIB`);
- the name of the library (`LIB`);
- the absolute path of the include directory, where the `.mod` files generated by the compiler will be written (`INTERFACES`);

The table below shows what compiler option to use for writing/reading `.mod` files in the directory specified in the Shell variable `INTERFACES` (which is defined in your `make.inc` file) for several well-known Fortran compilers:

Table 3: Compiler option for writing/reading `.mod` files in the `$INTERFACES` directory

Compiler	Command	Compiler options
GNU	gfortran	-J$(INTERFACES)
LLVM	flang	-module-dir $(INTERFACES)
AMD AOCC	flang	-J$(INTERFACES)
Intel Classic	ifort	-module $(INTERFACES)
Intel	ifx	-module $(INTERFACES)
PGI	pgfortran, pgf90, pgf95, pgf2003	-module $(INTERFACES)
NVIDIA	nvfortran	-module $(INTERFACES)
NAG	nagfor	-I$(INTERFACES) -mdir $(INTERFACES)
IBM XL	xlf90_r, xlf95_r, xlf2003_r	-I$(INTERFACES) -qmoddir=$(INTERFACES)

This command line option must be specified in the Shell variable `OPTS` defined in your `make.inc` file.

Two loader options are typically used for linking the object code of the STATPACK and, eventually, BLAS and LAPACK libraries, when creating an executable:

- **-l***name* causes the compiler to look for a library file named `lib`*name*`.a` and to link the executable to this library. To find this library file, the compiler searches sequentially through any directories named with the **-L** option explained below;
- **-L***dir* option lets you specify a (specific) directory for libraries specified with the **-l** option, before searching in the standard library directories `/lib` and `/usr/lib`.

Your compiler may have other options for specifying libraries, particularly if your UNIX system supports shared libraries and you want to use shared versions of the STATPACK, BLAS and LAPACK libraries.

Typically, all these loader options must be specified in the Shell variables `LBLAS`, `LLAPACK` and `LOADFLAGS` in your `make.inc` file. Look at the template `make.inc` files in the `$STATPACKDIR/makeincs` subdirectory for practical examples with different compilers.

Remember also when specifying these loader options in the Shell variable `LOADFLAGS` that UNIX linkers search for libraries in the order in which they occur on the command line and only resolve the references that are outstanding at the time when the library is searched. Therefore, the order of libraries and source/object files specified in `LOADFLAGS` can be critical and it is almost always a good idea to list first the STATPACK library and, secondly, only the LAPACK and BLAS libraries (or other libraries) in the Shell variable `LOADFLAGS` when compiling and linking STATPACK in order to avoid "Undefined" symbol messages during the loading or execution of an executable using the STATPACK routines.

Moreover, if STATPACK is built with OpenMP support, executables using the STATPACK library will be multi-threaded and the BLAS library eventually linked to STATPACK must be compiled thread-safe, as much as possible, in order to avoid unexpected errors at execution of your application. A simple way to achieve this, is to compile your BLAS library with OpenMP support. This will also ensure that OpenMP manages all the threads associated with your program, a feature, which is highly recommended to avoid performance problems at execution if your BLAS library is also multi-threaded. See the sections *OpenMP compilation* and *Parallel execution* for more details.

5) After you have built your `make.inc` file, the next step is to choose the real/complex kind types (**stnd** and **extd**), integer kind types (**i1b**, **i2b**, **i4b** and **i8b**) and logical kind type (**lgl**), which will be used in your version of the STATPACK library. These different kind types are merely named integer constants used by Fortran 95/2003 for defining parameterized real/complex, integer and logical types [Fortran]. All real/complex/integer/logical variables and constants used in STATPACK are defined in this way. See the Fortran program ex1_svd_cmp.F90 for an illustration.

A point that causes considerable nuisance in current Fortran libraries written in "old" Fortran like in BLAS and LAPACK is the need to maintain both single- and double-precision versions of exactly the same code. On the other hand, in Fortran 95/2003, there is no need to have separate versions of codes for single- and double-precision [Buckley:1994a] [Buckley:1994b] and STATPACK uses this possibility.

Most of the real/complex computations in STATPACK are done at the parameterized **stnd** real/complex precision. However, a few computations are preferably done at the higher (parameterized) precision **extd**. So, the kind type **extd** should be such that the underlying hardware will select a higher precision for kind **extd** than for kind **stnd**, if this is feasible. If a higher precision is not readily available, the same value may be used as for **stnd**.

Most current machines offer at least two precisions at the hardware level, very often three, and sometimes four. The decision about the correspondence between the parameterized **stnd** and **extd** kind types used in STATPACK and these different precisions at the hardware level is implemented by changing a single statement in the Fortran 90 *Select_Parameters* module, which is included in STATPACK.

Thus, for altering the precision of the computations performed in STATPACK, you need to edit the file `$STATPACKDIR/sources/Module_Select_Parameters.F90`, which contains the *Select_Parameters* module, and follow the instructions in the comments of this module.

Suffice to say here, that the user may select exactly the precision he wants for STATPACK by commenting/uncommenting lines in the *Select_Parameters* module, as in the following example:

```
!
! use The_Kinds, only : stnd=>sp,  extd=>dp
!
use The_Kinds, only : stnd=>dp,  extd=>qp
!
! use The_Kinds, only : stnd=>sp,  extd=>sp2
!
!use The_Kinds, only : stnd=>dp,  extd=>dp2
!
! use The_Kinds, only : stnd=>qp,  extd=>qp2
!
!
```

(continues on next page)

(continued from previous page)

```
! use The_Kinds, only : stnd=>low,    extd=>normal
!
! use The_Kinds, only : stnd=>normal,  extd=>extended
!
! use The_Kinds, only : stnd=>low,    extd=>low2
!
! use The_Kinds, only : stnd=>normal,  extd=>normal2
```

By simply ensuring that a leading '!' appears on all but exactly one of the preceding `use` statements in the *Select_Parameters* module, and then recompiling STATPACK, the precision of all the routines included in STATPACK can be altered. As an illustration, using the statement (e.g., uncommenting):

```
use The_Kinds, only : stnd=>sp,  extd=>dp
```

implies that the kind types **stnd** and **extd** used in STATPACK will now refer to single- and double-precision, respectively, after the recompilation of the code.

The symbolic names **sp**, **qp**, **qp**, ... used in the above example are defined in the *The_Kinds* module. The symbolic names **sp2**, **qp2**, **qp2**, ... are equivalent to **sp**, **qp**, **qp**, ..., respectively, but are used in *Select_Parameters* to avoid problems with some compilers, which do not allow that two different symbolic names in `use` statements may refer to the same entity in a Fortran 90 module.

The different choices for the kind type **stnd** (and also **extd**) are as follows. Selecting:

- **sp** kind for real/complex **stnd** data types in STATPACK requests to use the standard single precision available on all systems as the standard real or complex data type in STATPACK.
- **dp** kind for real/complex **stnd** data types in STATPACK requests to use the standard double precision available on all systems as the standard real or complex data type in STATPACK.
- **qp** kind for real/complex **stnd** data types in STATPACK requests to use the quadruple precision available on some systems as the standard data type in STATPACK (e.g., `qp = selected_real_kind( precision( 1.0d0 ) + 1 )` It is expected that this precision may not be available on all machines.
- **low** kind for real/complex **stnd** data types in STATPACK requests to use a real implementation "low" which provides at least 6 decimal digits of precision and an exponent range of at least 10^{+-35} as the standard real or complex data type in STATPACK. This would be suitable for low accuracy computations. It is expected that this precision will be available on all machines.
- **normal** kind for real/complex **stnd** data types in STATPACK requests to use a real implementation "normal" which provides at least 12 decimal digits of precision and an exponent range of at least 10^{+-50} as the standard real or complex data type in STATPACK. It is expected that this precision will be available on all machines.
- **extended** kind for real/complex **stnd** data types in STATPACK requests to use a real implementation "extended" which provides at least 20 decimal digits of precision and an exponent range of at least 10^{+-80} as the standard real or complex data type in STATPACK. It is expected that this precision may not be available on all machines.

Refer to the *The_Kinds* module for the exact definitions of all these different Fortran kind types.

By default (e.g., if you don't change anything in the *Select_Parameters* module, the parameterized **stnd** and **extd** real/complex kind types will both correspond to double-precision.

Both the integer (**i1b**, **i2b**, **i4b** and **i8b**) and logical (e.g., **lgl**) kind types available in STATPACK are parameterized in the same way. By default, the **i1b**, **i2b**, **i4b** and **i8b** integer types refer to 1-, 2-, 4- and 8-bytes integers and the **lgl** logical type refers to the standard logical type, defined as:

```
!
i1b   = selected_int_kind( 2  )
i2b   = selected_int_kind( 4  )
i4b   = selected_int_kind( 9  )
i8b   = selected_int_kind( 10 )
!
logic = kind( .true. )
```

See the definitions in the *The_Kinds* module for more informations. Again, these definitions can be altered by commenting/uncommenting statements in the *Select_Parameters* module.

Finally, note that you can use the Fortran program `test_kind.F90` (located in `$STATPACKDIR/sources`) to determine the available integer, real and logical kind types available on your computer at the hardware level and their properties. To compile and execute this program, simply execute the following **`make`** command:

```
$ make test_kind
```

in the main STATPACK directory (e.g., in `$STATPACKDIR`) and examine the standard output of the program on the screen. Note that if a real, integer or logical kind type defined in the *The_Kinds* module is not available on your computer, the named integer constant associated with it will be negative. Obviously, in that case, the associated kind type cannot be used in STATPACK.

Finally, as you can see in the *Select_Parameters* module, you can also modify manually other global parameters related to OpenMP compilations and cross-over between serial to vector algorithms (used only in the *Utilities* module) before proceeding to the compilation of the STATPACK library. This is useful if you need to tune STATPACK for improved efficiency on your platform. All the global control parameters of STATPACK are set in the *Select_Parameters* module.

6) For compiling and creating the STATPACK library, once you have built your own `make.inc` and customized appropriately the file `$STATPACKDIR/sources/Module_Select_Parameters.F90` with your choice for the real/complex kind types (**stnd** and **extd**), integer kind types (**i1b**, **i2b**, **i4b** and **i8b**) and logical kind type (**lgl**) used in STATPACK, execute the **`make`** command:

```
$ make lib
```

in the `$STATPACKDIR` or `$STATPACKDIR/sources` directory. If no errors are generated during this step, a static version of STATPACK library is now installed successfully on your computer (e.g., in the directory that you have specified in the Shell variable `DIRLIB` defined in your `make.inc` file). The library is called lib`$(LIB)`.a, where the Shell variable `LIB` is specified in your `make.inc` file.

All the public entities available in STATPACK are organized and grouped in Fortran 90 modules. The previous **`make`** command just compiles all the STATPACK modules taking into account the dependency between them. The compilation of each STATPACK module creates a `.mod` file and a `.o` file (note that some compilers do not create `.mod` files, however). The `.mod` file is used by the compiler at compile time to provide information about module contents. The `.o` file (if generated) contains the code of the STATPACK module procedures and must be specified when creating an executable file using the STATPACK procedures from this module.

All the `.o` files are subsequently joined together in the STATPACK library, which is located in the directory you specified in your `make.inc` file, after the successful completion of the **`make lib`** command.

Similarly, all the `.mod` files (if they exist) are located in the directory you specify in the Shell variable `INTERFACES` (again defined in your `make.inc`) or in the `$STATPACKDIR/sources` directory if this Shell variable `INTERFACES` is empty in your `make.inc` or if you didn't specify the appropriate compiler option to tell to the compiler where to write these `.mod` files.

On the other hand, if compilation errors occur at this step, please look at the section *Preprocessor cpp macros* and check if some cpp macros listed there can be useful for solving your compilation problem and must be added to the compilation options specified in your `make.inc` (e.g., in the Shell variable `OPTS`).

7) For creating a shared version of the STATPACK library (this is optional), execute the **make** command:

```
$ make dynlib
```

in the $STATPACKDIR directory. This shared library is installed at the same place (e.g., see the value of DIRLIB in your $STATPACKDIR/make.inc) than the static version of the STATPACK library. The shared library is called lib$(LIB).so, where LIB is specified in your $STATPACKDIR/make.inc file.

8) Next, if you want to make sure if STATPACK routines work or not, you may now run some testing programs, which are provided in the subdirectory $STATPACKDIR/tests of the STATPACK distribution. To run these installation tests, once you have built the library, you can enter the commands:

```
$ export OMP_NUM_THREADS=2     # If STATPACK has been built with OpenMP support
$ make test_install
```

The results of the tests are listed on the screen and are written in the file test_install.output, which is located in the subdirectory $STATPACKDIR/tests.

Examine the outputs for any obvious errors or problems. Notice, however, that these testing programs are not fully complete in this version of the software. In particular, the fact that some results of the tests are incorrect may only means that you have a (slight) loss of precision in some of the routines available in STATPACK. Additional checking is available with the **make** command:

```
$ make test_more
```

The results of these new tests are also listed on the screen and written in the file test_more.output, which is also located in the subdirectory $STATPACKDIR/tests. Examine again the output for any obvious errors or problems. Similarly, the fact that some results of these new tests are incorrect may only means that you have a (slight) loss of precision in some of the routines available in STATPACK and do not preclude the use of other STATPACK procedures.

9) Finally, to clean all the directories after building the library and running the test programs, enter the **make** command:

```
$ make clean
```

More details on the available commands for compiling and managing the STATPACK code can be found in the headers of the makefiles $STATPACKDIR/makefile and $STATPACKDIR/sources/makefile. As an illustration, you can use the following Makefile commands for managing the STATPACK source code and library (assuming that your current directory is $STATPACKDIR):

- If for some reasons you want to destroy the present version of the STATPACK library and the associated .mod files, enter the **make** command:

```
$ make clean_lib
```

- On many systems, you can also force the recompilation of all the source files (e.g., modules) in STATPACK by using the **make** command:

```
$ make lib FRC=FRC
```

- Finally, if you have set correctly the Shell variable CHECKFLAGS in your make.inc file, you can check the Fortran syntax in all the STATPACK modules, by entering the **make** command:

```
$ make check_all
```

The following sections provide more details on how to activate OpenMP support when compiling STATPACK, and on the UNIX preprocessor cpp macros, which can be used to compile/optimize STATPACK or solve some compilation problems with STATPACK.

2.2 OpenMP compilation

STATPACK is a parallel, multi-threaded library based on the OpenMP standard [openmp]. Support for at least OpenMP 3.0 API is requested for activation of OpenMP parallelism in this version of STATPACK.

In order to activate OpenMP parallelism in the STATPACK library, all compilers require you to use an appropriate compiler flag to turn on OpenMP compilation.

The table below shows what compiler option to use for several well-known Fortran compilers:

Table 4: OpenMP compilation flags

Compiler	Command	OpenMP flag
GNU	gfortran	-fopenmp
LLVM	flang	-fopenmp
Intel Classic	ifort	-fopenmp
Intel	ifx	-qopenmp
AMD AOCC	flang	-mp
PGI	pgfortran, pgf95, pgf90	-mp
NVIDIA	nvfortran	-mp
NAG	nagfor	-openmp
IBM XL	xlf90_r, xlf95_r, xlf2003_r	-qsmp=omp

Additional information on OpenMP compilation options provided by a large range of current Fortran compilers can be found at OpenMP compilers. You will also find several examples of how to activate OpenMP compilation for various compilers/platforms in the template `make.inc` files under the subdirectory `$STATPACKDIR/makeincs`.

How to activate parallelism when executing a program using STATPACK routines compiled with OpenMP support is described below in the section *Parallel execution*.

2.3 Preprocessor cpp macros

The STATPACK library uses the standard UNIX preprocessor, cpp, in order to allow some flexibility in the compilation of the STATPACK library and enhanced performance at execution. The cpp preprocessor is only used for conditional compilation of some parts of the STATPACK source code at the user option. This is typically done by defining some UNIX preprocessor cpp macros (e.g., variables governing conditional compilation in the STATPACK source files) at the compilation step of STATPACK, usually by specifying **-D**`name` as a compilation option, where `name` is a preprocessor cpp macro. Note that there is no space between **-D** and `name`. Each occurrence of **-D** defines a single macro and the **-D** option can appear many times on a command line.

Please note that your compiler may have other options for specifying UNIX preprocessor cpp macros (this is for example the case of the IBM XL Fortran compiler on IBM UNIX-like systems).

The following preprocessor cpp macros are currently used in the STATPACK source code and can be defined at compilation of STATPACK software in the Shell variable `OPTS` defined in your `make.inc` file:

- `_F2003` for activating the use of Fortran 2003 constructs and modules inside the STATPACK library. Please note that this includes the use of the intrinsic *IEEE_EXCEPTIONS*, *IEEE_ARITHMETIC* and *IEEE_FEATURES* modules available only in Fortran 2003 (see [Fortran] for further details). More specifically, the detection of NaNs in the STATPACK library will be done with the help of the **ieee_is_nan()** provided by the *IEEE_ARITHMETIC* module if the preprocessor cpp macro `_F2003` is activated.
- `_ISNAN` for activating the detection of NaNs in the STATPACK library by the intrinsic **isnan()** function, if your compiler supports this intrinsic function and the *IEEE_ARITHMETIC* module is not available with your compiler. The preprocessor cpp macro `_ISNAN` has no effect if the preprocessor cpp macro `_F2003` is also activated.
- `_SCALE` lets you select how numerical constants defined as integer powers of two are computed in the STATPACK library. If `_SCALE` is activated at compilation these numerical constants are computed with the **scale()** intrinsic function instead of direct exponentiation. This macro improves the portability of STATPACK for compilers which do not implement correctly exponentiation with integers or, alternatively, the **scale()** intrinsic function.
- `_NOT_DENORMALIZED` lets you inform STATPACK if denormalized floating point numbers are supported or not by your compiler. `_NOT_DENORMALIZED` can be activated at compilation for improving the accuracy of some scaling constants for computing Euclidean or Frobenius norms if you are sure that denormalized floating point numbers do not exist in the floating point arithmetic of your computer.
- `_BLAS` lets you activate the use of an optimized/multi-threaded BLAS library [blas] inside STATPACK as described in the section *Parallelism and BLAS*. Note that the name and path of this BLAS library must also be specified with the help of compiler/loader options in your `make.inc` file as described in the section *Basic installation*. Using the `_BLAS` cpp macro usually results in enhanced performance for most computing subroutines and functions available in STATPACK, especially if the `_DOT_PRODUCT` and `_MATMUL` cpp macros are also activated at the compilation of the STATPACK library. However, this will be true only if you links the STATPACK library with an optimized (and eventually multi-threaded) BLAS library such as GotoBLAS [gotoblas], OpenBLAS [openblas] or other vendor BLAS like Intel MKL [mkl]. Obviously, the `_BLAS` cpp macro can only be used if the parameterized **stnd** real/complex kind type you have selected corresponds to single- or double-precision on your platform.
- `_DOT_PRODUCT` tells to the Fortran compiler to replace each instance of the Fortran 90 intrinsic function, **dot_product()**, in the STATPACK source code by the corresponding STATPACK function, *`dot_product2()`*. Use the cpp macro `_DOT_PRODUCT`, if you suspect that the intrinsic Fortran 90 routines of your Fortran compiler are not optimized or efficient. If the cpp macro `_BLAS` is also defined, the BLAS subroutine *`dot()`* will be used. On many systems, the *`dot_product2()`* function is (much) faster than the intrinsic **dot_product()** function if the `_BLAS` cpp macro is also activated at the compilation of the STATPACK library.

- _MATMUL tells to the Fortran compiler to replace each instance of the Fortran 90 intrinsic function, **matmul()**, in the source code by the corresponding STATPACK function, *matmul2()*, which uses a blocked algorithm and is also multi-threaded when OpenMP is used. Use the cpp macro _MATMUL, if you suspect that the intrinsic Fortran 90 functions of your Fortran compiler are not optimized or efficient. If the cpp macro _BLAS is also defined, the BLAS subroutines *gemm()* and *gemv()* will be used instead of an OpenMP multi-threaded version of **matmul()**. On many systems, the *matmul2()* function is (much) faster than the intrinsic **matmul()** function, especially if the _BLAS cpp macro is also activated or if OpenMP is used at the compilation of the STATPACK library.
- _TRANSPOSE tells to the Fortran compiler to replace each instance of the Fortran 90 intrinsic function, **transpose()**, in the source code by the corresponding STATPACK function, *transpose2()*, which is multi-threaded when OpenMP is used. Use the cpp macro _TRANSPOSE, if you suspect that the intrinsic Fortran 90 functions of your Fortran compiler are not optimized or efficient. On many systems, the *transpose2()* function is (much) faster than the intrinsic **transpose()** function, especially if OpenMP is used at the compilation of the STATPACK library.
- _USE_GNU signals that the GNU gfortran compiler is used for compiling the STATPACK library. This includes only the activation of the _RANDOM_GFORTRAN cpp macro (described below) in this version of the STATPACK library.
- _USE_INTEL signals that the INTEL ifort compiler is used for compiling the STATPACK library. This includes only the activation of the _WHERE cpp macro (described below) in this version of the STATPACK library.
- _USE_NAGWARE signals that the NAG nagfor compiler is used for compiling the STATPACK library. This includes the activation of the _RANDOM_NAGWARE, _INTERNAL_PROC and _ORDERED cpp macros (described below) and the deactivation of some OpenMP directives/constructs in this version of the STATPACK library.
- _USE_PGI signals that the PGI pgfortran (or pgf90, pgf95, ...) or NVIDIA nvfortran compilers are used for compiling the STATPACK library. This includes only the deactivation of some OpenMP directives/constructs in this version of the STATPACK library.
- _ALLOC for allocating some local variables instead of of placing them on the stack in some subroutines and functions available in STATPACK.
- _WHERE for replacing Fortran 90 **where** constructs by **do** loops inside some OpenMP directives/constructs. This is useful for some compilers, like the INTEL ifort compiler, which does not allow **where** constructs inside some OpenMP directives in their OpenMP implementation.
- _ORDERED for deactivating OpenMP **DO** directives, which contain an OpenMP **ORDERED** clause. Useful for some compilers, which do not support (or do not implement correctly) the OpenMP **ORDERED** clause in their OpenMP implementation.
- _INTERNAL_PROC for deactivating OpenMP parallelization for sections of codes, which contain calls to internal procedures. This is useful for some fortran compilers like the MIPSpro f90 and the NAG nagfor compilers, which do not allow such possibility in their OpenMP implementation.
- _NOOPENMP2 for deactivating OpenMP parallelization in the tridiagonal eigensolvers and bidiagonal SVD solvers based on the implicit QR method available in the STATPACK library. Use only this cpp macro in case of OpenMP compilation problems with the *EIG_Procedures* and *SVD_Procedures* modules, included in the STATPACK library. Use of this cpp macro will result in a large performance degradation of the tridiagonal eigensolvers and bidiagonal SVD solvers based on the implicit QR method available in STATPACK.
- _NOOPENMP3 for deactivating OpenMP parallelization in the low-level subroutines and functions exported by the *Utilities* module available in STATPACK. By default, if OpenMP is used, these low-level routines are parallelized with OpenMP.
- _STAT for printing statistics on the number of deflated singular values in subroutines implementing the QR algorithm for bidiagonal matrices. Currently, this cpp macro is only used in the *SVD_Procedures* module available in STATPACK.

- _RANDOM_NOUNIX for signaling that the operating system is not UNIX. This cpp macro is only used in the *Random* module available in STATPACK.
- _RANDOM_WITH0 for generating real floating point numbers in the [0,1[interval instead of the]0,1[interval with the random generators available in the *Random* module included in STATPACK. This cpp macro is only used in the *Random* module available in STATPACK.
- _RANDOM_NOINT32 for signaling that 32 bit integers are not available with the compiler. This imposes some restrictions on the random generators available in STATPACK. This cpp macro is only used in the *Random* module available in STATPACK.
- _RANDOM_GFORTRAN for signaling that the UNIX integer **getpid()** function is considered as an intrinsic rather than an external procedure, as for the GNU gfortran compiler.
- _RANDOM_NAGWARE for signaling that the UNIX functions/subroutines, like the UNIX integer **getpid()** function, are part of the *f90_unix_env* Fortran 90 module when the NAG nagfor compiler is used. The cpp macro _RANDOM_NAGWARE takes care of this difference. This cpp macro is only used in the *Random* module available in STATPACK. Don't use the cpp macro _RANDOM_NAGWARE with other Fortran compilers since this will generate compilation errors for the *Random* module.

Examples of use of these preprocessor cpp macros for the compilation of STATPACK can be found in the template `make.inc` files under the subdirectory `$STATPACKDIR/makeincs`.

CHAPTER

THREE

STATPACK OVERVIEW

You will find several subdirectories under the STATPACK library directory: `sources`, `examples`, `tests`, `doc`, `interfaces`, `makeincs` and `myprograms`.

The content and use of these subdirectories are briefly described in this chapter.

3.1 sources directory

All the constants, variables, subroutines and functions available in the STATPACK library are organized and grouped in Fortran 90 modules. All the modules available in the library are located in the `sources` subdirectory of the STATPACK directory.

The tables below gives a brief overview of the different STATPACK modules:

Table 1: STATPACK modules and their contents

Module	Content
The_Kinds	Exports symbolic names for kinds of logical, integer and real/complex data types available at the hardware level
Select_Parameters	Selects and exports parameterized logical (**lgl**), integer (**i4b**, ...) and real/complex (**stnd** and **extd**) data types and other default parameters for the current version of STATPACK
Derived_Types	Defines and exports parameterized derived data types for sparse real and complex matrices of kind **stnd** and **extd**
Reals_Constants	Defines and exports names for almost all the literal real values of kind **stnd** and **extd** used in STATPACK
Num_Constants	Exports constants and functions for the machine dependent constants of real type of kind **stnd**

Table 2: STATPACK modules and their contents (cont.)

Module	Content
Logical_Constants	Defines and exports the logical constants `true` and `false` of kind **lgl**
Char_Constants	Exports character constants, strings and errors messages for routines available in STATPACK
Sort_Procedures	Exports sorting and ranking utilities for real and integer arrays of kind **stnd** and **i4b**
Print_Procedures	Exports printing utilities
String_Procedures	Exports utilities for manipulating strings and character data
Time_Procedures	Exports utilities for manipulating dates and time
Utilities	Exports simple computing routines (matrix multiplication, transposition, norms, . . .)
Random	Exports routines for random number and array generation, randomized linear algebra and related procedures
Giv_Procedures	Exports routines for computing and applying Givens rotations and reflections
Hous_Procedures	Exports routines for computing and applying Householder reflectors
QR_Procedures	Exports routines for computing QR, QRCP and LQ decompositions and related factorizations/computations
EIG_Procedures	Exports routines for solving the symmetric eigenvalues/eigenvectors problem and related factorizations/computations. Both standard and randomized routines are available
SVD_Procedures	Exports routines for computing the Singular Value Decomposition of a matrix and related factorizations/computations. Both standard and randomized routines are available
Lin_Procedures	Exports routines for solving linear systems, computing the inverse and determinant of a matrix and related decompositions (LU, Cholesky, . . .)

Table 3: STATPACK modules and their contents (cont.)

Module	Content
LLSQ_Procedures	Exports routines for solving linear least square problems and related computations
Prob_Procedures	Exports routines for probability distribution functions and their inverses
Stat_Procedures	Exports routines for univariate statistical computations
Mul_Stat_Procedures	Exports routines for multivariate statistical computations
FFT_Procedures	Exports routines for (fast) Fourier transform computations
Time_Series_Procedures	Exports routines for time series analysis
BLAS_interfaces	Exports generic interfaces for selected routines in the BLAS library
Lapack_interfaces	Exports generic interfaces for selected routines in the LAPACK library
Statpack	Exports all the public entities available in STATPACK

The content of each STATPACK module and the purpose of the public entities exported by this module are fully described in the chapter *STATPACK reference manual*. For more information on the use of a specific routine available in STATPACK, you must consult the reference section for the module exporting the routine or the appropriate STATPACK manual for this module.

The `sources` subdirectory also contains the Fortran programs `alphabet.f90`, `test_kind.F90` and `mach_char.F90`:

- `alphabet.f90` can be used for a simple check of your Fortran compiler. You can use this Fortran program as soon as you have built your own `make.inc` file, as described in the section *Basic installation*. To compile and execute this program, simply execute the following **`make`** command:

```
$ make alphabet
```

  in the main STATPACK directory (e.g., in `$STATPACKDIR`) or `sources` subdirectory. This program will simply display the ASCII characters on the standard output of the program (e.g., the screen).

- `test_kind.f90` can be used to test and display informations on the different real/complex, integer and logical kind types available on your platform. You can use this Fortran program as soon as you have built your own `make.inc` file, as described in the section *Basic installation*. To compile and execute this program, simply execute the following **`make`** command:

```
$ make test_kind
```

in the main STATPACK directory (e.g., in `$STATPACKDIR`) or `sources` subdirectory. Informations about the available integer, real and logical kind types available on your computer at the hardware level and their properties will be displayed on the standard output of the program (e.g., the screen).

- `mach_char.f90` can be used to test and display detailed numerical properties of a specific real/complex kind type available at the hardware level on your computer. You can use this Fortran program as soon as you have built your own `make.inc` file, as described in the section *Basic installation*. To compile and execute this program, simply execute the following **`make`** command:

```
$ make mach_char
```

in the main STATPACK directory (e.g., in `$STATPACKDIR`) or `sources` subdirectory. By default, the program will determine the parameters of the floating-point arithmetic system for double-precision real/complex data, which is available on all platforms, and will display these parameters on the screen. In order to obtain information about machine-specific parameters for another real/complex kind type, it is necessary to edit the file `mach_char.f90` and to comment out all but one of the following `use` statements, before compiling the program:

```
!
!   use The_Kinds, only : stnd=>sp
!
use The_Kinds, only : stnd=>dp
!
!   use The_Kinds, only : stnd=>qp
!
!   use The_Kinds, only : stnd=>low
!
!   use The_Kinds, only : stnd=>normal
!
!   use The_Kinds, only : stnd=>extended
```

The following **`make`** commands are available in the `sources` subdirectory to manage the STATPACK source code:

- To create or update the library, enter the **`make`** command:

```
$ make lib
```

Alternatively, the **`make`** command:

```
$ make
```

without any arguments creates also the STATPACK library. The library is called lib`$(LIB)`.a, where `LIB` is specified in your `$STATPACKDIR/make.inc` file.

- To create a shared version of the library, enter the **`make`** command:

```
$ make dynlib
```

The shared library is installed in the directory that you have specified in `DIRLIB` defined in your `$STATPACKDIR/make.inc` file.

- On some systems, you can force the source files to be recompiled by entering the **`make`** command:

```
$ make lib FRC=FRC
```

- To check the fortran syntax in a single module, for example the Hous_Procedures module in the `Module_Hous_Procedures.F90`, enter the **make** command:

```
$ make Module_Hous_Procedures.check
```

  This **make** command will work properly only if you have defined properly the Shell variable `CHECKFLAGS` in your `$STATPACKDIR/make.inc` file.

- To check the fortran syntax in all STATPACK modules, enter the **make** command:

```
$ make check_all
```

  This **make** command will work properly only if you have defined properly the Shell variable `CHECKFLAGS` in your `$STATPACKDIR/make.inc` file.

- Finally, to clean the `sources` subdirectory after building the library, enter the **make** command:

```
$ make clean
```

3.2 tests directory

The subdirectory `tests` contains all the testing programs for the routines available in the STATPACK library.

The name of the test programs is determined by the STATPACK routine, which is tested by the program. As an illustration, the test program `test_svd_cmp.F90` is the test program for the *svd_cmp()* STATPACK subroutine.

Instructions for running these test programs can be found in the header of the makefile in this subdirectory.

The following **make** commands are available in the `tests` subdirectory to compile/execute/manage the STATPACK test programs:

- To see the list of all the test programs, enter the **make** command:

```
$ make list
```

- To compile and run a particular test program in this list, enter the **make** command (for example):

```
$ make test_svd_cmp
```

  The program and the results are printed on the screen and stored in the current directory (e.g., `tests`) in the file named `test_svd_cmp.output`.

- To run all the installation tests, enter the **make** command:

```
$ make test_install
```

  Alternatively, the **make** command:

```
$ make
```

  without any arguments runs also all the installation tests. The results of the tests are stored in the `tests` directory in the file named `test_install.output`.

- Additional tests are available for some routines and can be performed by entering the **make** command:

```
$ make test_more
```

  The results of these tests are stored in the `tests` directory in the file named `test_more.output`.

- To see the list of all (test) programs which can be compiled in this directory, enter the **make** command:

```
$ make list_compil
```

- To compile a particular test program in this list, enter the **make** command (for example):

```
$ make test_svd_cmp.compil
```

  The executable is generated in the current directory and is called `a.out`.

- To clean the `tests` directory, enter the **make** command:

```
$ make clean
```

3.3 examples directory

Many sample Fortran 95/2003 programs that illustrate the use of STATPACK routines are available in the `examples` subdirectory of the STATPACK directory. The name of the programs is determined by the STATPACK routine, whose use is illustrated by the program. As two illustrations, the program `ex1_svd_cmp.F90` is the first example for the *`svd_cmp()`* STATPACK subroutine and the program `ex1_lapack_ormtr.F90` is the first example of the generic interface *`ormtr()`* for the LAPACK subroutines **sormtr()**, **dormtr()**, **cormtr()** and **zormtr()** (see the description of the *Lapack_interfaces* module for more details).

Instructions for compiling and running these example programs can be found in the header of the makefile in this subdirectory.

The following **make** commands are available in the `examples` subdirectory to compile/execute/manage the STATPACK example programs:

- To see the list of all the example programs, enter the **make** command:

```
$ make list
```

- To compile and run a particular example or program in this list, enter the **make** command (for example):

```
$ make ex1_svd_cmp
```

  The program and the results are printed on the screen and stored in the current directory (e.g., `examples`) in the file named `ex1_svd_cmp.output`.

- To see the list of all (example) programs which can be compiled in this directory, enter the **make** command:

```
$ make list_compil
```

- To compile a particular program in this list, enter the **make** command (for example):

```
$ make ex1_svd_cmp.compil
```

  The executable is generated in the current directory and is called `a.out`

- To clean the `examples` directory, enter the **make** command:

```
$ make clean
```

You can also put your own Fortran 95/2003 programs using the STATPACK library in the `examples` subdirectory and use the above **make** commands to compile easily these programs without the need to create your own makefile. However, it it better to use the `myprograms` subdirectory described below for this purpose.

3.4 doc directory

The subdirectory `doc` contains some STATPACK documentation in different formats (e.g., pdf and html).

3.5 interfaces directory

The empty subdirectory `interfaces` can be used, at the user option, to store the `.mod` files generated by the compiler on some systems (e.g., NAGWare, RS6K/IBM, LINUX, Mac OSX machines). See the section *Basic installation* for more details on the possible use of this subdirectory.

3.6 makeincs directory

The subdirectory `makeincs` contains examples and templates of `make.inc` files for different compilers/machines, which can be useful to build your own `make.inc` file. See the section *Basic installation* for more details.

Templates are currently provided for the *gfortran* (`make.inc.GNU`), *ifort* (`make.inc.INTEL`), *pgfortran* (`make.inc.PGI`), *xlf95* (`make.inc.IBM`) and *nagfor* (`make.inc.NAG`) compilers.

Some `make.inc` examples really used on some machines are also provided.

3.7 myprograms directory

You can put your own Fortran 95/2003 programs using the STATPACK library in the subdirectory `myprograms`. See the following chapter *Using the STATPACK library* for more details.

CHAPTER

FOUR

USING THE STATPACK LIBRARY

This chapter describes how to compile and run programs that use the STATPACK library, and introduces its main conventions.

4.1 Example program

Note, first, that in order to use one of the STATPACK parameterized kind types, routines or constants in your program, you must include an `use Statpack` statement in your Fortran program, like:

```
use Statpack, only: stnd, i4b, solve_lin
```

The module *Statpack*, used in the example above, exports all the constants, routines and functions publicly available in STATPACK. Alternatively, you can also refer directly to the Fortran module, which contains the routine you want to use in your `use` statement, but it is much less convenient since you must remember for each STATPACK entity, the module which contains this entity.

The following complete program illustrates the use of the STATPACK function *`solve_lin()`* for solving a system of linear equations.

After, using an appropriate `use Statpack` statement for all the needed STATPACK entities, the code first allocates a square matrix `mat`, a solution vector `x`, a right hand side vector `y` and two working vectors, `x2` and `x2`, of real kind **stnd**.

Next, the matrix `mat` and solution vector `x` are filled up with real random numbers of kind **stnd** and the corresponding right hand side vector `y` is generated by a matrix multiplication using the intrinsic **matmul()** function.

The resulting linear system composed by the coefficient matrix `mat` and the right hand side `y` is solved with the help of the *`solve_lin()`* STATPACK function to recover the solution vector `x` in the vector `x2`.

The accuracy of the solution is checked and the program finally prints some information collected during the process (e.g., error and timing of the computations).

```
program example_using_statpack
!
!
! Purpose
! =======
!
!   This program is intended to demonstrate the use of fonction SOLVE_LIN
!   in STATPACK.
!
! LATEST REVISION : 13/06/2018
!
```

(continues on next page)

(continued from previous page)

```
! ================================================================================
!
!
! USED MODULES
! ============
!
    use Statpack, only : i4b, stnd, solve_lin, merror, allocate_error
!
!
! STRONG TYPING IMPOSED
! =====================
!
    implicit none
!
!
! PARAMETERS
! ==========
!
! prtunit IS THE PRINTING UNIT, n IS THE SIZE OF THE LINEAR SYSTEM
!
    integer(i4b), parameter :: prtunit=6, n=4000
!
    character(len=*), parameter :: name_proc='Example of solve_lin'
!
!
! SPECIFICATIONS FOR VARIABLES
! ============================
!
    real(stnd)                                :: err, eps, elapsed_time
    real(stnd), dimension(:,:), allocatable :: a
    real(stnd), dimension(:),   allocatable :: b, x, x2, res
!
    integer :: iok, istart, iend, irate
!
!
! EXECUTABLE STATEMENTS
! =====================
!
!
!   EXAMPLE 1 : REAL MATRIX AND ONE RIGHT HAND-SIDE.
!
!   SET THE REQUIRED PRECISION OF THE RESULTS.
!
    eps = sqrt( epsilon( err ) )
!
!   ALLOCATE WORK ARRAYS.
!
    allocate( a(n,n), b(n), x(n), x2(n), res(n), stat=iok )
!
    if ( iok/=0 ) then
        call merror( name_proc//allocate_error )
    end if
!
!   GENERATE A n-by-n RANDOM DATA MATRIX a .
!
    call random_number( a )
!
```

(continues on next page)

(continued from previous page)

```
!   GENERATE A n RANDOM SOLUTION VECTOR x .
!
    call random_number( x )
!
!   COMPUTE THE MATRIX-VECTOR PRODUCT b = a*x .
!
    b(:n) = matmul( a(:n,:n), x(:n) )
!
!   START TIMING THE COMPUTATIONS.
!
    call system_clock( count=istart, count_rate=irate )
!
!   COMPUTE THE SOLUTION VECTOR FOR LINEAR SYSTEM
!
!            a*x = b .
!
!   BY COMPUTING THE LU DECOMPOSITION WITH PARTIAL PIVOTING AND
!   IMPLICIT ROW SCALING OF MATRIX a WITH FUNCTION solve_lin.
!   ARGUMENTS a AND b ARE NOT MODIFIED BY THE FUNCTION.
!
    x2(:n) = solve_lin( a(:n,:n), b(:n) )
!
!   STOP THE TIMER.
!
    call system_clock( count=iend )
!
    elapsed_time = real( iend - istart, stnd )/real( irate, stnd )
!
!   CHECK THE RESULTS FOR SMALL RESIDUALS.
!
    res(:n) = x2(:n) - x(:n)
    err     = sum( abs(res(:n)) ) / sum( abs(x(:n)) )
!
!   DEALLOCATE WORK ARRAYS.
!
    deallocate( a, b, x, x2, res )
!
!   CHECK THE RESULTS FOR SMALL RESIDUALS.
!
    if ( err<=eps ) then
        write (prtunit,*) name_proc//' is correct'
    else
        write (prtunit,*) name_proc//' is incorrect'
    end if
!
    write (prtunit,*)
    write (*,'(a,i5,a,0pd12.4,a)')    &
    'The elapsed time for computing the solution of a linear real system of size ',&
    n, ' is', elapsed_time, ' seconds'
!
!
! END OF PROGRAM example_using_statpack
! =====================================
!
end program example_using_statpack
```

Assuming that this sample program is in the file `example_using_statpack.f90`, the steps to compile and link

this sample program are detailed in the following section.

4.2 Compiling and linking

The simplest way is to copy the source file `example_using_statpack.f90` (or your own program) in the `$STATPACKDIR/myprograms` directory. For illustration purpose, a copy of `example_using_statpack.f90` is already stored in this directory.

To see the programs, which can be compiled and/or executed in the `$STATPACKDIR/myprograms` directory (the files must have the suffix .f90, .F90, .f95, or .F95), enter the **`make`** command:

```
$ make list
```

in this directory. The program *example_using_statpack* will appear as a target in the list. If you want to compile and execute directly the program *example_using_statpack*, just enter the **`make`** command:

```
$ make example_using_statpack
```

This command creates the executable `example_using_statpack.out` in the current directory, based on the informations given in your `$STATPACKDIR/make.inc` file, and executes it.

Alternatively, if you just want to compile this program, just enter the **`make`** command:

```
$ make example_using_statpack.compil
```

This also creates the executable `example_using_statpack.out` in the current directory, but without executing it. Moreover, the exact command used for creating the executable is printed on the screen and you can use this command as a model to compile and link any program using your STATPACK library outside from the `$STATPACKDIR/myprograms` directory.

To see the list of programs which can be compiled (e.g., the files with the suffix .f90, .F90, .f95 or .F95), enter the **`make`** command:

```
$ make list_compil
```

Finally, to clean the `$STATPACKDIR/myprograms` directory, enter the **`make`** command:

```
$ make clean
```

4.3 Shared libraries

To run a program linked with the shared version of the STATPACK library, the operating system must be able to locate the corresponding `.so` file at runtime. This shared version of the library can be created with the **`make dynlib`** command under the `$STATPACKDIR` directory (see the section *Basic installation* for more details).

If a shared library cannot be found (for example the STATPACK library), the following error will occur:

```
$ ./example_using_statpack.out
./example_using_statpack.out: error while loading shared libraries:
lib_statpack.so: cannot open shared object file: No such file or directory
```

Typically, this means that the Shell variable `LOADFLAGS` in your `$STATPACKDIR/make.inc` file is not correctly defined and you must correct it. To avoid this error, either modify your `$STATPACKDIR/make.inc` file or define the Shell variable `LD_LIBRARY_PATH` to include the directory where the library is installed.

For example, in the Bourne Shell (e.g., */bin/sh* or */bin/bash*), the library search path can be updated with the following command:

```
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/statpack/library
```

where `/path/to/statpack/library` is the directory where the STATPACK library is installed. In the C-shell (e.g., */bin/csh* or */bin/tcsh*) the equivalent command is:

```
% setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:/path/to/statpack/library
```

The standard prompt for the C-shell in the example above is the percent character **%**, and should not be typed as part of the command. To save retyping these commands each session, they can be placed in an individual or system-wide login file.

Finally, remember that the shared library dependencies of an executable can be listed with the **`ldd`** command (on a Unix system):

```
$ ldd ./example_using_statpack.out
linux-vdso.so.1 =>  (0x00007ffec16ed000)
lib_statpack.so => /usr/home/terray/statpack2/lib_statpack.so (0x00002b7193d05000)
libopenblas.so.0 => /usr/home/terray/lib-OpenBLAS-0.2.20-icc-ifort-bulldozer/lib/
↪libopenblas.so.0 (0x00002b71946f5000)
libifport.so.5 => /opt/intel/15.0.6.233/composer_xe_2015.6.233/compiler/lib/intel64/
↪libifport.so.5 (0x00002b7195975000)
libifcoremt.so.5 => /opt/intel/15.0.6.233/composer_xe_2015.6.233/compiler/lib/intel64/
↪libifcoremt.so.5 (0x00002b7195ba5000)
libimf.so => /opt/intel/15.0.6.233/composer_xe_2015.6.233/compiler/lib/intel64/libimf.
↪so (0x00002b7195f0d000)
libsvml.so => /opt/intel/15.0.6.233/composer_xe_2015.6.233/compiler/lib/intel64/
↪libsvml.so (0x00002b71963cd000)
libm.so.6 => /lib64/libm.so.6 (0x00002b71972c5000)
libiomp5.so => /opt/intel/15.0.6.233/composer_xe_2015.6.233/compiler/lib/intel64/
↪libiomp5.so (0x00002b71975cd000)
libintlc.so.5 => /opt/intel/15.0.6.233/composer_xe_2015.6.233/compiler/lib/intel64/
↪libintlc.so.5 (0x00002b7197915000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x00002b7197b75000)
libc.so.6 => /lib64/libc.so.6 (0x00002b7197d95000)
libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00002b7198165000)
libdl.so.2 => /lib64/libdl.so.2 (0x00002b719837d000)
libirng.so => /opt/intel/15.0.6.233/composer_xe_2015.6.233/compiler/lib/intel64/
↪libirng.so (0x00002b7198585000)
/lib64/ld-linux-x86-64.so.2 (0x000055e74098c000)
```

4.4 Parallel execution

Users may request a specific number of OpenMP threads to distribute the work done by an application using the STATPACK library, when OpenMP support has been activated at compilation of STATPACK.

As a general rule, don't request more OpenMP threads than the number of processors available on your machine (excluding also processors used for hyperthreading), this will result in large loss of performance. Keep also in mind that the efficiency of shared-memory parallelism as implemented in STATPACK with OpenMP also depends heavily on the workload of your shared-memory computer at runtime.

More generally, threading performance of an application using STATPACK will depend on a variety of factors including the compiler, the version of the OpenMP library, the processor type, the number of cores, the amount of

available memory, whether hyperthreading is enabled and the mix of applications that are executing concurrently with the application.

At the simplest level, the number of OpenMP threads used by an OpenMP multi-threaded application can be controlled by setting the `OMP_NUM_THREADS` OpenMP environment variable to the desired number of threads and the number of threads will be the same throughout the execution of the application. The `OMP_NUM_THREADS` OpenMP environment variable must be defined before the execution of the multi-threaded application to activate OpenMP parallelism.

Setting OpenMP environment variables is done the same way you set any other environment variables, and depends upon which Shell you use:

Table 1: Setting the number of OpenMP threads to be used

Shell	Command line
csh/tcsh	setenv OMP_NUM_THREADS 8
sh/bash	export OMP_NUM_THREADS=8

In some cases, an OpenMP program will perform better if its OpenMP threads are bound to processors/cores (this is called "thread affinity", "thread binding" or "processor affinity") because this can result in better cache utilization, thereby reducing costly memory accesses. OpenMP version 3.1 API provides an environment variable to turn processor binding "on" or "off". For example, to turn "on" thread binding you can use:

```
$ export OMP_PROC_BIND=TRUE   #if you are using a sh/bash Shell
```

Keep also in mind, that the OpenMP standard does not specify how much stack space an OpenMP thread should have. Consequently, implementations will differ in the default thread stack size and the default thread stack size can be easily exhausted for moderate/large applications on some systems. Threads that exceed their stack allocation may give a segmentation fault or the application may continue to run while data is being corrupted. If your OpenMP environment supports the OpenMP 3.0 `OMP_STACKSIZE` environment variable, you can use it to set the thread stack size prior to program execution. For example:

```
$ export OMP_STACKSIZE=10M   #if you are using a sh/bash Shell
$ export OMP_STACKSIZE=3000k #if you are using a sh/bash Shell
```

More generally, the run-time behaviour of an OpenMP multi-threaded application is also determined by setting some other OpenMP environment variables (e.g., `OMP_NESTED`, `OMP_MAX_ACTIVE_LEVELS` or `OMP_DYNAMIC` for example) just before the execution of the application. See the official OpenMP documentation available at OpenMP or the more friendly tutorial OpenMP Environment Variables for more details and examples about OpenMP environment variables you can use.

All this management of the OpenMP threads can also be controlled and done inside your Fortran program with the help of the OpenMP API run-time library routines [openmp]. Consult the relevant information here OpenMP Run Time Library

Note, in particular, that the STATPACK routines may use OpenMP nested parallelism if the `OMP_NESTED` variable is set to `TRUE` or if the OpenMP run-time routine **omp_set_nested()** is used in your program to enable nested parallelism (e.g., calling the OpenMP subroutine **omp_set_nested()** with the value `.true.` will enabled nested parallelism after the call at runtime).

Keep also in mind that, starting with the OpenMP 5.0 API, the use of both the `OMP_NESTED` variable and **omp_set_nested()** subroutine are deprecated and must be replaced by the use of the `OMP_MAX_ACTIVE_LEVELS` OpenMP variable and the **omp_set_max_active_levels()** run-time subroutine, already available in the OpenMP 3.0 API. See OpenMP Environment Variables for more details on the use of the `OMP_MAX_ACTIVE_LEVELS` variable

and OpenMP Run Time Library for the use of the **omp_set_max_active_levels()** run-time subroutine to turn off/on nested OpenMP parallelism and the level of nested parallelism in Fortran programs.

However, the usage of OpenMP nested parallelism is not recommended if you have compiled the STATPACK library with BLAS support and you have linked with a multi-threaded version of BLAS, such as [gotoblas], [openblas] or vendor BLAS like Intel MKL [mkl]. In such cases, it is strongly recommended to first desactivate OpenMP nested parallelism before executing of your application by using first the command:

```
$ export OMP_NESTED=FALSE    #if you are using a sh/bash Shell
```

or the command:

```
$ export OMP_MAX_ACTIVE_LEVELS=1    #if you are using a sh/bash Shell
```

and also to let OpenMP controls the multi-threading in the BLAS library, if possible.

In the case of OpenBLAS [openblas] or GotoBLAS [gotoblas], this can be done by using the makefile `USE_OPENMP=1` option when compiling OpenBLAS or GotoBLAS. Consult the OpenBLAS manual for more details [openblas].

On the other hand, if your OpenBLAS or GotoBLAS library has already been compiled with multi-threading enabled, but no support for OpenMP (this is the default setting), it is strongly recommended to make sure that the number of threads used by these libraries is equal to one when STATPACK routines are called. Otherwise, OpenMP will not control the multi-threading in the BLAS routines called by the STATPACK routines and this will likely results in large loss of performance. To do this, use a command like (for OpenBLAS):

```
$ export OPENBLAS_NUM_THREADS=1    #if you are using a sh/bash Shell
```

or (for GotoBLAS):

```
$ export GOTO_NUM_THREADS=1    #if you are using a sh/bash Shell
```

before executing your application. In both cases, OpenBLAS or GotoBLAS will use only one thread throughout the execution of your program/application. Executing `call openblas_set_num_threads(1)` or `call gotoblas_set_num_threads(1)` right before a call to a STATPACK routine will do also. These calls have higher priority than the `OPENBLAS_NUM_THREADS` and `GOTOBLAS_NUM_THREADS` environment variables, respectively, and allow a finer control over the parallelism in your application (see the OpenBLAS or GotoBLAS documentation for more details).

Similarly, for Intel MKL [mkl], it is better to let OpenMP controls the multi-threading in the MKL BLAS. This can be done simply by undefining the Shell variable `MKL_NUM_THREADS`, which controls the number of threads (cores) for the Intel MKL BLAS library, before executing your application:

```
$ unset MKL_NUM_THREADS    #if you are using sh/bash Shell
```

4.5 Using long integers

If you are using huge data arrays (i.e., if indexing exceeds `2^32-1`), it may be useful or even mandatory to define the **i4b** integer type used in STATPACK library as 64 bit.

In order to do this, the first step is to select the appropriate parameterized **i4b** integer kind type used in STATPACK. By default, the **i1b**, **i2b**, **i4b** and **i8b** integer types used in STATPACK refer to 1-, 2-, 4- and 8-bytes integers, respectively; but these default correspondances can be altered simply by commenting/uncommenting lines in the *Select_Parameters* module, as in the following example:

```
!
!   use The_Kinds, only : i1b      , i2b      , i4b      , i8b
!
!   use The_Kinds, only : i1b=>i4b1, i2b=>i4b2, i4b      , i8b
!
!   use The_Kinds, only : i1b=>i4b1, i2b=>i4b2, i4b      , i8b=>i4b8
!
!   use The_Kinds, only : i1b=>i8b1, i2b=>i8b2, i4b=>i8b4, i8b
!
use The_Kinds, only : i1b      , i2b      , i4b=>i8b4, i8b
```

The last statement redefines all the **i4b** integers used in STATPACK as **i8b** integers. Once you have customized appropriately the file `$STATPACKDIR/sources/Module_Select_Parameters.F90` with these choices for the integer kind types (**i1b**, **i2b**, **i4b** and **i8b**) used in STATPACK, execute the **`make`** command:

```
$ make lib
```

in the `$STATPACKDIR` or `$STATPACKDIR/sources` directory to recompile the full STATPACK library with 64-bit integers. See the section *Basic installation* for more details.

Note, however, that if you want to use your new 64-bit integer STATPACK library with BLAS support and you plan to link your 64-bit integer STATPACK library with a (multi-threaded) version of BLAS, such as [gotoblas], [openblas] or vendor BLAS like Intel MKL [mkl], both the STATPACK and BLAS libraries should also be compiled with the `-i8` (for the INTEL ifort compiler) or the `-fdefault-integer-8` (for the GNU gfortran compiler) flag (this compiler option defines automatically default integers used in a Fortran program as 64 bit); otherwise the generic interfaces for the BLAS subroutines defined in the module *BLAS_interfaces* will not work properly. In the case of Intel MKL BLAS [mkl], this means that you must link STATPACK with the ilp64 version of the Intel MKL library, which defines default integers as 64 bit (the standard lp64 version of Intel MKL library assumes that default integers are standard 32 bit). In the case of OpenBLAS [openblas] or GotoBLAS [gotoblas], this can be done by using the makefile `INTERFACE64=1` option when compiling OpenBLAS or GotoBLAS. Consult the OpenBLAS manual for more details [openblas].

These considerations apply also if you are planning to use both the STATPACK and LAPACK libraries [lapack] in your Fortran code with the help of the generic interfaces for the LAPACK subroutines defined in the module *Lapack_interfaces*.

CHAPTER

FIVE

STATPACK REFERENCE MANUAL

5.1 Introduction

Constants, variables, subroutines and functions available in the STATPACK library are organized and grouped in Fortran 90 modules. All the modules available in the STATPACK library are located in the `$STATPACKDIR/sources` subdirectory of the main STATPACK directory. To use these STATPACK modules in a Fortran program, you must have previously compiled the STATPACK library as described in the chapter *Installation*.

The content of each module is listed in this reference chapter with one or two description paragraphs of the purpose of the module and the list of constants, subroutines and functions publicly available in the module. For more information on the use of a specific routine, you must follow the links to the specific documentation of this routine or consult the STATPACK manual for the appropriate module.

In order to use one of the STATPACK routines or constants described below, you must include an appropriate `use Statpack` statement in your Fortran program, like:

```
use Statpack, only: svd_cmp
```

The module *Statpack*, used in the example above, exports all the constants, routines and functions publicly available in STATPACK. Alternatively, you can also refer directly to the Fortran module, which contains the routine you want to use in your `use` statement, like:

```
use SVD_Procedures, only: svd_cmp
```

See the Fortran program ex1_svd_cmp.F90 for a working example of subroutine *svd_cmp()* and the STATPACK library for performing a Singular Value Decomposition (SVD) of a real matrix.

In this reference section, for each routine publicly available in STATPACK, we give its calling sequence (or the different calling sequences if this routine is generic) and the list of dummy arguments to the routine. For example, the different calling sequences of the generic *quick_sort()* subroutine available in the module *Sort_Procedures*, which can be used to sort integer or real arrays, are as follow:

```
call quick_sort( list(:p) ,              ascending=ascending )   ! list is a␣
↪real array of size p
call quick_sort( list(:p) , order(:p) , ascending=ascending )   ! list is a␣
↪real array of size p
call quick_sort( list(:p) ,              ascending=ascending )   ! list is an␣
↪integer array of size p
call quick_sort( list(:p) , order(:p) , ascending=ascending )   ! list is an␣
↪integer array of size p
```

In the above calling sequences, all the possible dummy arguments (and forms of the generic routine) are listed. The dimensions of the dummy array arguments are also indicated, following the Fortran90 notation, and the dependencies

between the dimensions of the different dummy array arguments are also indicated when this is possible. The mandatory dummy arguments are listed first by using an ordinary positional argument list and the optional dummy arguments are listed after by using a keyword argument list. For example, in the second form of the generic *quick_sort()* subroutine, the dummy array arguments *LIST* and *ORDER* are mandatory and their sizes must match. On the other hand, *ASCENDING* is an optional dummy argument.

For more information on the purpose of the routine and the possible arguments (including their types, sizes or shapes), you must follow the links to the specific documentation of this routine (e.g., click on the name of the subroutine, which is underlined in this reference chapter) or consult the STATPACK manual for the appropriate module.

5.2 MODULE The_Kinds

Module *The_Kinds* exports symbolic names for kinds of logical, integer, real or complex types available on the computer.

Here is the list of the useful symbolic names exported by module *The_Kinds*:

```
!
! SYMBOLIC NAME FOR DEFAULT KIND OF LOGICAL:
!
integer, parameter :: logic = kind( .true. )
!
! SYMBOLIC NAMES FOR KIND TYPES OF LOGICAL:
!
integer, parameter :: logic0 = 0
integer, parameter :: logic1 = 1
integer, parameter :: logic2 = 2
integer, parameter :: logic4 = 4
!
! SYMBOLIC NAMES FOR KIND TYPES OF 1-, 2-, 4- and 8-BYTES INTEGERS:
!
integer, parameter :: i1b = selected_int_kind( 2  )
integer, parameter :: i2b = selected_int_kind( 4  )
integer, parameter :: i4b = selected_int_kind( 9  )
integer, parameter :: i8b = selected_int_kind( 10 )
!
! SYMBOLIC NAMES FOR KIND TYPES OF SINGLE-, DOUBLE- and QUADRUPLE-PRECISION REAL
! AND COMPLEX NUMBERS:
!
integer, parameter :: sp = kind( 1.0   )
integer, parameter :: dp = kind( 1.0d0 )
integer, parameter :: qp = selected_real_kind( precision( 1.0d0 ) + 1 )
!
! THE qp KIND TYPE MAY NOT BE AVAILABLE ON YOUR COMPUTER.
!
! PRECISION SPECIFICATIONS FOR REAL AND COMPLEX COMPUTATIONS:
!
integer, parameter :: low = selected_real_kind( 6, 35 )
integer, parameter :: normal = selected_real_kind( 12, 50 )
integer, parameter :: extended = selected_real_kind( 20, 80 )
!
! THESE PRECISION SPECIFICATIONS REQUEST, RESPECTIVELY, 6, 12, 20 DECIMAL DIGITS OF
! PRECISION AND AN EXPONENT RANGE OF AT LEAST 10 ^ +- 35, 10 ^ +- 50 AND 10 ^ +- 80.
! THE extended PRECISION MAY NOT BE AVAILABLE ON YOUR COMPUTER.
!
```

To know the available kind types and precisions on your computer, you can use the program `test_kind.F90` , e.g., simply execute the **`make`** command:

```
$ make test_kind
```

in the main STATPACK directory.

The choice between these different kind types and precisions for compiling a version of STATPACK is done in the module *Select_Parameters* (see the source file `Module_Select_Parameters.F90`).

5.3 MODULE Select_Parameters

Module *Select_Parameters* provides a convenient way of selecting:

- the precision (e.g., the **stnd** and **extd** real kind types) required for the computations in STATPACK
- the size of integer (e.g., the **i1b**, **i2b**, **i4b** and **i8b** integer kind types) or logical (e.g., the **lgl** logical kind type) constants and variables used in STATPACK
- the default printing unit
- the block size for blocked algorithms in modules *Utilities*, *Lin_Procedures*, *FFT_Procedures*, *QR_Procedures*, *Random*, *Eig_Procedures* and *SVD_Procedures*
- the parameters for OpenMP compilation
- the parameters for crossover from serial to parallel algorithms for routines in module *Utilities*
- the parameters for the STATPACK testing programs
- the location of the urandom device on your system if it exists, which can used to seed the STATPACK random generators in module *Random* in an optimal fashion

In order to change the default kind types and make your own choice for the above parameters, you must edit the source file `Module_Select_Parameters.F90` and follow the instructions in this file, as detailed below:

```
!
! USED MODULES
! ============
!
! ------------------------------------------------------------------------
! By simply ensuring that a leading '!' appears on all but exactly one !
! of the following use statements, and then recompiling all routines,  !
! the size of integer variables can be changed.                        !
! No harm will be done if short integers are made the same as i4b or   !
! i8b integers.                                                        !
! ------------------------------------------------------------------------
!
use The_Kinds, only : i1b      , i2b      , i4b      , i8b
!
! use The_Kinds, only : i1b=>i4b1, i2b=>i4b2, i4b      , i8b
!
! use The_Kinds, only : i1b=>i4b1, i2b=>i4b2, i4b      , i8b=>i4b8
!
! use The_Kinds, only : i1b=>i8b1, i2b=>i8b2, i4b=>i8b4, i8b
!
! use The_Kinds, only : i1b      , i2b      , i4b=>i8b4, i8b
!
!
```

(continues on next page)

(continued from previous page)

```
! ------------------------------------------------------------------------
! By simply ensuring that a leading '!' appears on all but exactly one  !
! of the following use statements, and then recompiling all routines,   !
! the precision of an entire real or complex computation can be altered.!
!                                                                        !
! A few computations are preferably done in higher precision 'extd'. So,!
! the kind type 'extd' should be such that the underlying hardware will !
! select a higher precision for kind 'extd' than for kind 'stnd', if    !
! this is feasible.  If a higher precision is not readily available,    !
! the same value may be used as for 'stnd'.                             !
! ------------------------------------------------------------------------
!
! use The_Kinds, only : stnd=>sp,  extd=>dp
!
! use The_Kinds, only : stnd=>dp,  extd=>qp
!
! use The_Kinds, only : stnd=>sp,  extd=>sp2
!
use The_Kinds, only : stnd=>dp,  extd=>dp2
!
! use The_Kinds, only : stnd=>qp,  extd=>qp2
!
! use The_Kinds, only : stnd=>low,     extd=>normal
!
! use The_Kinds, only : stnd=>normal,  extd=>extended
!
! use The_Kinds, only : stnd=>low,     extd=>low2
!
! use The_Kinds, only : stnd=>normal,  extd=>normal2
!
! ------------------------------------------------------------------------
! By simply ensuring that a leading '!' appears on all but exactly one !
! of the following use statements, and then recompiling all routines,   !
! the size of logical variables can be changed.                         !
! ------------------------------------------------------------------------
!
use The_Kinds, only : lgl=>logic
!
! use The_Kinds, only : lgl=>logic0
!
! use The_Kinds, only : lgl=>logic1
!
! use The_Kinds, only : lgl=>logic2
!
! use The_Kinds, only : lgl=>logic4
!
```

Similarly, the default values for the other parameters specified in module *Select_Parameters* can be changed or tuned for your computer at your convenience. For example, tuning specifically the default block sizes of the linear algebra routines included in STATPACK for your computer will lead to significant speed improvements in most cases. See the file `Module_Select_Parameters.F90` for more details.

In order to use one of these kind types or other parameters, you must include an appropriate `use Select_Parameters` or `use Statpack` statement in your Fortran program, like:

```
use Select_Parameters, only: lgl, i4b, stnd
```

or:

```
use Statpack, only: lgl, i4b, stnd
```

5.4 MODULE Derived_Types

Module *Derived_Types* exports derived data types for sparse real and complex matrices of kind **stnd** and **extd**. These two kind types are defined in module *Select_Parameters*.

The available derived data types are defined as follow:

```
!
! DERIVED DATA TYPES FOR SPARSE MATRICES WITH KIND stnd AND extd
! ==============================================================
!
type sprs2_stnd
    integer(i4b) :: n, len
    real(stnd),   dimension(:), pointer :: val
    integer(i4b), dimension(:), pointer :: irow
    integer(i4b), dimension(:), pointer :: jcol
end type sprs2_stnd
!
type sprs2_extd
    integer(i4b) :: n, len
    real(extd),   dimension(:), pointer :: val
    integer(i4b), dimension(:), pointer :: irow
    integer(i4b), dimension(:), pointer :: jcol
end type sprs2_extd
!
type sprs2_stndc
    integer(i4b) :: n, len
    complex(stnd), dimension(:), pointer :: val
    integer(i4b),  dimension(:), pointer :: irow
    integer(i4b),  dimension(:), pointer :: jcol
end type sprs2_stndc
!
type sprs2_extdc
    integer(i4b) :: n, len
    complex(extd), dimension(:), pointer :: val
    integer(i4b),  dimension(:), pointer :: irow
    integer(i4b),  dimension(:), pointer :: jcol
end type sprs2_extdc
```

5.5 MODULE Reals_Constants

Module *Reals_Constants* provides names for almost all the literal real values of kind **stnd** and **extd** used in STATPACK.

The real/complex kind types **stnd** and **extd** are defined in module *Select_Parameters*.

By using only real values as defined within the module *Reals_Constants*, all conversion problems associated with the precision of real literal values in STATPACK can be totally avoided.

Note, finally, that the code of module *Reals_Constants* is in the source file `Modules_Constants.F90`

Here is the list of the most useful public constants exported by module *Reals_Constants*:

```
!
! REAL CONSTANTS AT PRECISION stnd.
!
!   DIGITS
!
real(stnd), parameter ::       &
    zero    = 0,               &
    one     = 1,               &
    two     = 2,               &
    three   = 3,               &
    four    = 4,               &
    five    = 5,               &
    six     = 6,               &
    seven   = 7,               &
    eight   = 8,               &
    nine    = 9,               &
    ten     =10
!
!   TENTHS
!
real(stnd), parameter ::       &
    c0_1    = 0.1_stnd,        &
    c0_2    = 0.2_stnd,        &
    c0_3    = 0.3_stnd,        &
    c0_4    = 0.4_stnd,        &
    c0_5    = 0.5_stnd,        &
    c0_6    = 0.6_stnd,        &
    c0_7    = 0.7_stnd,        &
    c0_8    = 0.8_stnd,        &
    c0_9    = 0.9_stnd
!
!   RECIPROCALS
!
real(stnd), parameter ::       &
    tenth   = 0.1_stnd,        &
    ninth   = one/nine,        &
    eighth  = 0.125_stnd,      &
    seventh = one/seven,       &
    sixth   = one/six,         &
    fifth   = 0.2_stnd,        &
    quarter = 0.25_stnd,       &
    third   = one/three,       &
    half    = 0.5_stnd
!
!   INTEGRAL VALUES TO 99
!
real(stnd), parameter ::                                                  &
    c10     = 10,          c40     = 40,          c70     = 70,          &
    c11     = 11,          c41     = 41,          c71     = 71,          &
    c12     = 12,          c42     = 42,          c72     = 72,          &
    c13     = 13,          c43     = 43,          c73     = 73,          &
    c14     = 14,          c44     = 44,          c74     = 74,          &
    c15     = 15,          c45     = 45,          c75     = 75,          &
    c16     = 16,          c46     = 46,          c76     = 76,          &
    c17     = 17,          c47     = 47,          c77     = 77,          &
    c18     = 18,          c48     = 48,          c78     = 78,          &
```

(continues on next page)

(continued from previous page)

```
    c19      = 19,            c49      = 49,            c79      = 79,    &
    c20      = 20,            c50      = 50,            c80      = 80,    &
    c21      = 21,            c51      = 51,            c81      = 81,    &
    c22      = 22,            c52      = 52,            c82      = 82,    &
    c23      = 23,            c53      = 53,            c83      = 83,    &
    c24      = 24,            c54      = 54,            c84      = 84,    &
    c25      = 25,            c55      = 55,            c85      = 85,    &
    c26      = 26,            c56      = 56,            c86      = 86,    &
    c27      = 27,            c57      = 57,            c87      = 87,    &
    c28      = 28,            c58      = 58,            c88      = 88,    &
    c29      = 29,            c59      = 59,            c89      = 89,    &
    c30      = 30,            c60      = 60,            c90      = 90,    &
    c31      = 31,            c61      = 61,            c91      = 91,    &
    c32      = 32,            c62      = 62,            c92      = 92,    &
    c33      = 33,            c63      = 63,            c93      = 93,    &
    c34      = 34,            c64      = 64,            c94      = 94,    &
    c35      = 35,            c65      = 65,            c95      = 95,    &
    c36      = 36,            c66      = 66,            c96      = 96,    &
    c37      = 37,            c67      = 67,            c97      = 97,    &
    c38      = 38,            c68      = 68,            c98      = 98,    &
    c39      = 39,            c69      = 69,            c99      = 99
!
!   MISCELLANEOUS INTEGRAL VALUES
!
real(stnd), parameter ::       &
    c100     = 100,            &
    c120     = 120,            &
    c180     = 180,            &
    c200     = 200,            &
    c256     = 256,            &
    c300     = 300,            &
    c360     = 360,            &
    c400     = 400,            &
    c500     = 500,            &
    c600     = 600,            &
    c681     = 681,            &
    c700     = 700,            &
    c800     = 800,            &
    c900     = 900,            &
    c991     = 991,            &
    c1000    = 1000,           &
    c1162    = 1162,           &
    c2324    = 2324,           &
    c2000    = 2000,           &
    c3000    = 3000,           &
    c4000    = 4000,           &
    c5000    = 5000,           &
    c10000   = 10000,          &
    c20700   = 20700,          &
    c40000   = 40000
!
!   FREQUENTLY USED MATHEMATICAL CONSTANT
!
real(stnd), parameter ::                                                  &
    pi       = 3.14159265358979323846264338327950288419716939937510_stnd,&
    pio2     = 1.57079632679489661923132169163975144209858_stnd,         &
    twopi    = 6.28318530717958647692528676655900576839_stnd,            &
```

(continues on next page)

(continued from previous page)

```
    sqrt2     = 1.41421356237309504880168872420969807856967_stnd,        &
    euler     = 0.57721566490153286060651209008240243104422_stnd
!
!
! REAL CONSTANTS AT PRECISION extd.
!
!   DIGITS
!
real(extd), parameter ::            &
    zero_extd     = 0,              &
    one_extd      = 1,              &
    two_extd      = 2,              &
    three_extd    = 3,              &
    four_extd     = 4,              &
    five_extd     = 5,              &
    six_extd      = 6,              &
    seven_extd    = 7,              &
    eight_extd    = 8,              &
    nine_extd     = 9,              &
    ten_extd      =10
!
!   FREQUENTLY USED MATHEMATICAL CONSTANTS
!
real(extd), parameter ::                                                  &
    pi_extd        = 3.14159265358979323846264338327950288419716939937751_extd,&
    pio2_extd      = 1.57079632679489661923132169163975144209858_extd,    &
    twopi_extd     = 6.28318530717958647692528676655900576839 4_extd,     &
    sqrt2_extd     = 1.41421356237309504880168872420969807856967_extd,    &
    euler_extd     = 0.57721566490153286060651209008240243104422_extd,    &
    lnsqrt2pi_extd = 0.9189385332046727_extd
```

Other public real constants of kind **stnd** and **extd** exported by module *Reals_Constants* are used in specific STATPACK routines.

5.6 MODULE Logical_Constants

Module *Logical_Constants* exports logical constants of kind **lgl**.

The logical kind type **lgl** is defined in module *Select_Parameters*.

By using logical values as defined in this module, all problems associated with the conversion of logical literal values in STATPACK can be totally avoided.

Note, finally, that the code of module *Logical_Constants* is in the source file `Modules_Constants.F90`

Here is the list of the public constants exported by module *Logical_Constants*:

```
!
! LOGICAL CONSTANTS OF KIND lgl.
!
logical(lgl), parameter ::    &
    true  = .true._lgl,       &
    false = .false._lgl
```

In order to use one of these constants, you must include an appropriate `use Logical_Constants` or `use Statpack` statement in your Fortran program, like:

```
use Logical_Constants, only: true
```

or:

```
use Statpack, only: true
```

5.7 MODULE Char_Constants

Module *Char_Constants* Char_Constants exports character constants, strings and errors messages for routines available in STATPACK.

The code of module *Char_Constants* is in the source file `Modules_Constants.F90`

Here is the list of the useful public character constants and strings exported by module *Char_Constants*:

```
!
! NAMES FOR COMMON CHARACTERS.
!
character(len=1), parameter ::     &
    ampersand   = achar(38)  ,     &
    apostrophe  = achar(39)  ,     &
    atSign      = achar(64)  ,     &
    backslash   = achar(92)  ,     &
    backquote   = achar(96)  ,     &
    bang        = achar(33)  ,     &
    blank       = achar(32)  ,     &
    caret       = achar(94)  ,     &
    cbrace      = achar(125) ,     &
    cbracket    = achar(93)  ,     &
    cparen      = achar(41)  ,     &
    colon       = achar(58)  ,     &
    comma       = achar(44)  ,     &
    dash        = achar(45)  ,     &
    dollar      = achar(36)  ,     &
    equals      = achar(61)  ,     &
    exclamation = achar(33)  ,     &
    greaterthan = achar(62)  ,     &
    hash        = achar(35)
!
character(len=1), parameter ::     &
    lessthan    = achar(60)  ,     &
    minus       = achar(45)  ,     &
    obrace      = achar(123) ,     &
    obracket    = achar(91)  ,     &
    oparen      = achar(40)  ,     &
    percent     = achar(37)  ,     &
    period      = achar(46)  ,     &
    plus        = achar(43)  ,     &
    quesmark    = achar(63)  ,     &
    quote       = achar(34)  ,     &
    semicolon   = achar(59)  ,     &
    slash       = achar(47)  ,     &
    star        = achar(42)  ,     &
    tilde       = achar(126) ,     &
    vertBar     = achar(124) ,     &
```

(continues on next page)

(continued from previous page)

```
    underscore  = achar(95)
!
! NAME FOR NULL STRING.
!
character(len=0), parameter :: null = ''
!
! NAMES FOR ASCII COMMAND CHARACTERS.
!
character(len=1), parameter ::     &
    bell = achar(7) ,              & ! BELL
    bs   = achar(8) ,              & ! BACK SPACE
    ht   = achar(9) ,              & ! HORIZONTAL TABULATION
    lf   = achar(10) ,             & ! LINE FEED
    vt   = achar(11) ,             & ! VERTICAL TABULATION
    ff   = achar(12) ,             & ! FORM FEED
    cr   = achar(13) ,             & ! CARRIAGE RETURN
    so   = achar(14) ,             & ! SHIFT OUT
    si   = achar(15) ,             & ! SHIFT IN
    esc  = achar(27) ,             & ! ESCAPE
    del  = achar(127)                ! DELETE
!
! ERROR MESSAGE FOR allocate STATEMENT .
!
character(len=*), parameter ::                                    &
allocate_error  = ' : problem in attempt to allocate memory !'
```

In order to use one of these constants or strings, you must include an appropriate `use Char_Constants` or `use Statpack` statement in your Fortran program, like:

```
use Char_Constants, only: underscore
```

or :

```
use Statpack, only: underscore
```

Other public strings exported by module *Char_Constants* concern error messages for STATPACK routines.

5.8 MODULE Num_Constants

Module *Num_Constants* exports constants and functions for the machine dependent constants of real type of kind **stnd**.

Routines for identifying and manipulating NaNs (e.g., Not A Number) for real (or complex) data of kind **stnd** are also provided.

These special routines will use the intrinsic modules *IEEE_EXCEPTIONS*, *IEEE_ARITHMETIC* and *IEEE_FEATURES* , if the compiler provides support for the IEEE standard with this kind **stnd** and if the cpp macro `_F2003` is activated at compilation of the STATPACK library. If support for the IEEE standard is not available, but the intrinsic function **isnan()** is available with your compiler, you can alternatively activate the cpp macro `_ISNAN` at compilation of the STATPACK library so that this intrinsic function will be used inside the STATPACK library to check for the presence of NaNs. However, if the cpp macro `_F2003` is activated, the cpp macro `_ISNAN` has no effect.

The real/complex kind type **stnd** is defined in module *Select_Parameters*.

Note, finally, that the code of module *Num_Constants* is in the source file `Modules_Constants.F90`

Here is the list of the public constants exported by module *Num_Constants*:

```
!
! VALUES OF MACHINE NUMERIC CHARACTERISTICS
! WITH INTRINSIC VALUES IN FORTRAN90.
!
integer(i4b), parameter ::                &
    maxexp      = maxexponent(unitrnd), & ! LARGEST EXPONENT BEFORE OVERFLOW
    minexp      = minexponent(unitrnd), & ! MINIMUM EXPONENT BEFORE (GRADUAL)␣
↪UNDERFLOW
    base        = radix(unitrnd),       & ! BASE OF THE MACHINE
    nbasedigits = digits(unitrnd),      & ! NUMBER OF (base) DIGITS IN THE MANTISSA
    decprec     = precision(unitrnd),   & ! NUMBER OF EQUIVALENT DECIMAL DIGITS IN␣
↪THE MANTISSA
    decexpr     = range(unitrnd)          ! EQUIVALENT DECIMAL EXPONENT RANGE
!
real(stnd), parameter ::                   &
    machmaxexp      = maxexp,              & ! LARGEST EXPONENT BEFORE OVERFLOW
    machminexp      = minexp,              & ! MINIMUM EXPONENT BEFORE (GRADUAL)␣
↪UNDERFLOW
    machbase        = base,                & ! BASE OF THE MACHINE
    machnbasedigits = nbasedigits,         & ! NUMBER OF (base) DIGITS IN THE MANTISSA
    machdecprec     = decprec,             & ! NUMBER OF EQUIVALENT DECIMAL DIGITS IN␣
↪THE MANTISSA
    machdecexpr     = decexpr,             & ! EQUIVALENT DECIMAL EXPONENT RANGE
    machulp         = epsilon(unitrnd),    & ! MACHINE PRECISION   :  base**(1-
↪nbasedigits)
    macheps         = machulp*half,        & ! MACHINE EPSILON, ASSUMING ROUNDING : ␣
↪half*machulp
    machtiny        = tiny(unitrnd),       & ! UNDERFLOW THRESHOLD :  base**(minexp-1)
    machhuge        = huge(unitrnd)          ! OVERFLOW THRESHOLD  : (base**maxexp)*(1-
↪base**(-nbasedigits))
!
! SCALING CONSTANTS FOR COMPUTING EUCLIDEAN OR FROBENIUS NORMS OF REAL OR COMPLEX␣
↪ARRAYS AND
! SCALING SUBROUTINES, SEE https://doi.org/10.1145/3061665 FOR MORE DETAILS.
!
real(stnd), parameter ::                                          &
    safmin      = machbase**max(minexp-1_i4b,1_i4b-maxexp), & ! SAFE MINIMUM :␣
↪base**max(minexp-1,1-maxexp)
    safmax      = one/safmin,                               & ! SAFE MAXIMUM
    machsmlnum = safmin/machulp,                            & ! SCALED MINIMUM
    machbignum = safmax*machulp,                            & ! SCALED MAXIMUM
    rtmin       = sqrt(machsmlnum),                         & ! SQUARE ROOT OF SCALED␣
↪MINIMUM
    rtmax       = sqrt(machbignum)                            ! SQUARE ROOT OF SCALED␣
↪MAXIMUM
!
! SCALING CONSTANTS FOR COMPUTING EUCLIDEAN OR FROBENIUS NORMS OF REAL OR COMPLEX␣
↪ARRAYS
! WITH THE BLUE'S ALGORITHM (e.g., https://doi.org/10.1145/355769.355771). NOTE THAT␣
↪lcs HAS
! BEEN CORRECTED TO SCALE DENORMALIZED NUMBERS CORRECTLY, SEE https://doi.org/10.1145/
↪3061665
! FOR MORE DETAILS.
!
integer, parameter ::  lcb = ceiling(real(minexp-1_i4b,kind=stnd)*half),            &
                       ucb = floor(real(maxexp-nbasedigits+1_i4b,kind=stnd)*half),  &
```

(continues on next page)

(continued from previous page)

```
                        lcs = floor(real(minexp-nbasedigits,kind=stnd)*half),        &
                        ucs = ceiling(real(maxexp+nbasedigits-1_i4b,kind=stnd)*half)
!
real(stnd), parameter  :: tbig = machbase**(ucb),  &
                          sbig = machbase**(-ucs), &
                          ubig = machbase**(ucs)
!
real(stnd), parameter  :: tsml = machbase**(lcb),  &
                          ssml = machbase**(-lcs), &
                          usml = machbase**(lcs)
```

In order to use one of these constants or one of the routines listed below, you must include an appropriate `use Num_Constants` or `use Statpack` statement in your Fortran program, like:

```
use Num_Constants, only: base
```

or :

```
use Statpack, only: base
```

Here is the list of the public routines exported by module *Num_Constants*:

lamch()

Purpose:

lamch() determines machine parameters for the real/complex parameterized precision **stnd** as defined in the *Select_Parameters* module.

The routine is based on the routine DLAMCH() in LAPACK.

Synopsis:

```
x = lamch( cmach )
```

mach()

Purpose:

mach() is intended to determine the parameters and the properties of the floating-point arithmetic system specified with the real/complex parameterized precision **stnd**, as defined in the *Select_Parameters* module.

This subroutines is based on the MACHAR() subroutine developped by [Cody:1988] and DLAMCH() in LAPACK.

See [Malcolm:1972] [Gentleman_Marovich:1974] [Cody:1988] for more details.

Synopsis:

```
call mach( basedigits=basedigits , irnd=irnd , iuflow=iuflow , igrd=igrd ,␣
↪iexp=iexp , ifloat=ifloat , &
           expepspos=expepspos , expepsneg=expepsneg , minexpbase=minexpbase ,
↪ maxexpbase=maxexpbase , &
           epspos=epspos , epsneg=epsneg , epsilpos=epsilpos ,␣
↪epsilneg=epsilneg , rndunit=rndunit     )
```

test_ieee()

Purpose:

test_ieee() try to determine if the computer follows the IEEE standard 754 for binary floating-point arithmetic for the real/complex parameterized precision **stnd** defined in the *Select_Parameters* module.

test_ieee() returns `true` if the computer seems to follow the IEEE standard 754 and `false` otherwise.

If the compiler follows the Fortran 2003 standard and the cpp macro _F2003 is activated at compilation of the STATPACK library, the facilities provided by the *IEEE_ARITHMETIC* intrinsic module are used to determine if the computer follows the IEEE standard 754 for binary floating-point arithmetic. Otherwise results from [Cody_Coonen:1993] are used.

Synopsis:

```
test = test_ieee( )
```

test_nan()

Purpose:

test_nan() returns `true` if NaNs exist, and `false` otherwise.

If the compiler follows the Fortran 2003 standard and the cpp macro _F2003 is activated at compilation of the STATPACK library, the facilities provided by the *IEEE_ARITHMETIC* intrinsic module are used to determine if NaNs exist as defined in the IEEE standard 754 for binary floating-point arithmetic.

Otherwise, **test_nan()** exploits the IEEE requirement that NaNs compare as unequal to all values, including themselves [Cody_Coonen:1993].

Synopsis:

```
test = test_nan( )
```

is_nan()

Purpose:

is_nan() returns `true` if the real scalar *X* is a NaN or if the real array *X* contains a NaN, and `false` otherwise.

If the compiler follows the Fortran 2003 standard and the cpp macro _F2003 is activated at compilation of the STATPACK library, the facilities provided by the *IEEE_ARITHMETIC* intrinsic module are used to determine if NaNs are present as defined in the IEEE standard 754 for binary floating-point arithmetic. Alternatively, if the cpp macro _F2003 is not activated, but the compiler supports the intrinsic function **isnan()**, this function will used to detect NaNs if the cpp macro _ISNAN is activated at compilation of the STATPACK library.

Otherwise, **is_nan()** exploits the IEEE requirement that NaNs compare as unequal to all values, including themselves [Cody_Coonen:1993].

Synopsis:

```
test = is_nan( x       )
test = is_nan( x(:)    )
test = is_nan( x(:,:) )
```

replace_nan()

Purpose:

replace_nan() replaces:

- the real scalar *X* with the scalar *MISSING*, if *X* is a NaN on input;
- the NaNs in the input real array *X* with the scalar *MISSING*.

If the compiler follows the Fortran 2003 standard and the cpp macro _F2003 is activated at compilation of the STATPACK library, the facilities provided by the *IEEE_ARITHMETIC* intrinsic module are used to determine if NaNs are present as defined in the IEEE standard 754 for binary floating-point arithmetic. Alternatively, if the cpp macro _F2003 is not activated, but the compiler supports the intrinsic function **isnan()**, this function will used to detect NaNs if the cpp macro _ISNAN is activated at compilation of the STATPACK library.

Otherwise, **replace_nan()** exploits the IEEE requirement that NaNs compare as unequal to all values, including themselves [Cody_Coonen:1993].

Synopsis:

```
call replace_nan( x      , missing )
call replace_nan( x(:)   , missing )
call replace_nan( x(:,:) , missing )
```

nan()

Purpose:

nan() returns as a scalar function, the bit pattern corresponding to a quiet NaN in the IEEE standard 754 for binary floating-point arithmetic if the machine recognizes NaNs or the maximum floating point number of kind **stnd** otherwise (e.g. `huge(1._stnd)`.

If the compiler follows the Fortran 2003 standard, the facilities provided by the *IEEE_ARITHMETIC* module are used to create a quiet NaN as defined in the IEEE standard 754 for binary floating-point arithmetic.

Otherwise, the routine exploits the IEEE requirement that NaNs compare as unequal to all values, including themselves [Cody_Coonen:1993].

Finally, NAN returns the maximum floating point number of kind **stnd**, if the computer does not follow the IEEE standard 754 for binary floating-point arithmetic.

Synopsis:

```
x = nan( )
```

true_nan()

Purpose:

true_nan() returns as a scalar function, the bit pattern corresponding to a quiet NaN in the IEEE standard 754 for binary floating-point arithmetic, independently of the fact that the computer follows or not the IEEE standard 754 for binary floating-point arithmetic for the real/complex datat of kind **stnd**.

Synopsis:

```
x = true_nan( )
```

5.9 MODULE Sort_Procedures

Module *Sort_Procedures* exports routines for sorting vectors and matrices of real type of kind **stnd** and integer type of kind **i4b**. These two kind types are defined in module *Select_Parameters*.

Routines are provided for sorting data, both directly and indirectly (using an index) and are mostly based on the *Quicksort* algorithm [Knuth:1997] [Sedgewick:1998], which is a recursive $O(N \log N)$ algorithm. Routines are also provided to compute the ranks of the elements of a real or integer vector. The *rank* of an element is its order in the sorted data. The vector of ranks is the inverse permutation of the *index* permutation, which gives the order of the elements in their original sequence after sorting.

In order to use one of these routines, you must include an appropriate `use Sort_Procedures` or `use Statpack` statement in your Fortran program, like:

```
use Sort_Procedures, only: tri_insert
```

or :

```
use Statpack, only: tri_insert
```

Here is the list of the public routines exported by module *Sort_Procedures*:

tri_insert()

Purpose:

tri_insert() sort the integer or real array *LIST* into ascending numerical order, by straight insertion. *LIST* is replaced on output by its sorted rearrangement and the optional vector integer argument *ORDER* gives the positions of the elements in the original order.

Synopsis:

```
call tri_insert( list(:n)             )   ! list is a real array
call tri_insert( list(:n) , order(:n) )   ! list is a real array
call tri_insert( list(:n)             )   ! list is an integer array
call tri_insert( list(:n) , order(:n) )   ! list is an integer array
```

quick_sort()

Purpose:

quick_sort() sort the integer or real array *LIST* into ascending or descending numerical order, by the Quicksort algorithm [Knuth:1997] [Sedgewick:1998]. *LIST* is replaced on output by its sorted rearrangement and the optional vector integer argument *ORDER* gives the positions of the elements in the original order.

Synopsis:

```
call quick_sort( list(:n) ,             ascending=ascending )   ! list is a␣
↪real array
call quick_sort( list(:n) , order(:n) , ascending=ascending )   ! list is a␣
↪real array
call quick_sort( list(:n) ,             ascending=ascending )   ! list is an␣
↪integer array
call quick_sort( list(:n) , order(:n) , ascending=ascending )   ! list is an␣
↪integer array
```

Examples:

ex1_quick_sort.F90

do_index()

purpose:

do_index() indexes an integer or real array *LIST*, i.e., outputs the array index *INDEX* of length `n` such that `LIST(INDEX(j))` is in ascending order for `j=1, 2, ..., n`.

The input array *LIST* is not changed.

Synopsis:

```
call do_index( list(:n) , index(:n) ) ! list is a real array
call do_index( list(:n) , index(:n) ) ! list is an integer array
```

Examples:

ex1_do_index.F90

rank()

purpose:

Given an integer index array `index` as output from the routine *do_index()*, **rank()** returns a same-size integer vector *rank*, the corresponding array of ranks.

Synopsis:

```
vec(:n) = rank( index(:n) )
```

reorder()

purpose:

Given an integer index array `index` as output from the routine *do_index()*, **reorder()** makes the corresponding rearrangement of the same-size integer or real array `slave`.

The rearrangement is performed by means of the index array `index` and the logical input argument *ascending*.

Synopsis:

```
call reorder( index(:n) , slave(:n),    ascending=ascending )   ! slave is a␣
↪real array
call reorder( index(:n) , slave(:p,:n), ascending=ascending )   ! slave is a␣
↪real array
call reorder( index(:n) , slave(:n),    ascending=ascending )   ! slave is an␣
↪integer array
call reorder( index(:n) , slave(:p,:n), ascending=ascending )   ! slave is an␣
↪integer array
call reorder( index(:n) , slave(:n),    ascending=ascending )   ! slave is a␣
↪complex array
call reorder( index(:n) , slave(:p,:n), ascending=ascending )   ! slave is a␣
↪complex array
```

Examples:

ex1_reorder.F90

5.10 MODULE Print_Procedures

Module *Print_Procedures* exports constants and routines for printing vectors, matrices and results from other STATPACK routines.

Here is the list of the public constants exported by module Print_Procedures:

```
!
! DEFAULT VALUES FOR PRINTING.
!
integer(i4b), parameter :: defline   = 80 ,   & ! DEFAULT LINE SIZE
                           defindent = 0  ,   & ! DEFAULT INDENTATION
                           defw      = 12 ,   & ! DEFAULT WIDTH FOR EDIT DESCRIPTOR
                           defd      = 6  ,   & ! DEFAULT NUMBER OF SIGNIFICANT DIGITS
                           defs      = 3        ! DEFAULT NUMBER OF SPACES BETWEEN␣
↪ENTRIES
```

In order to use one of these constants or one of the routines listed below, you must include an appropriate `use Print_Procedures` or `use Statpack` statement in your Fortran program, like:

```
use Print_Procedures, only: defline
```

or :

```
use Statpack, only: defline
```

Here is the list of the public routines exported by module *Print_Procedures*:

enter_proc()

purpose:

Upon entering a procedure, **enter_proc()** can be called. It would skips two lines and outputs a message that the routine, identified by the string argument *STRING*, was entered. If the optional argument *LEVEL* is present, the message *STRING* is prepended by *LEVEL* blanks.

Synopsis:

```
call enter_proc( string , level=level , prt_unit=prt_unit )
```

leave_proc()

purpose:

leave_proc() is the *opposite* of routine *enter_proc()*. It should be called just before leaving a routine. The exit message *STRING* is output on the unit *PRT_UNIT* (or the default unit `defunit` defined in the *Select_Parameters* module if *PRT_UNIT* is absent) and two lines are skipped.

Optionally, the output message *STRING* is prepended by *LEVEL* blanks.

Synopsis:

```
call leave_proc( string , level=level , prt_unit=prt_unit )
```

entering()

purpose:

Upon entering a procedure, **entering()** can be called. It will return a prefix string suitable for indenting output lines from the procedure. It takes the given character argument *STRING* and prepends *LEVEL* blanks, followed by a `[`, and appends the character `]`.

For example, if `string = "hi"` and `level = 7`, it would return `_____[hi]`. *LEVEL* is then also incremented by `2`.

Trailing blanks in *STRING* are removed. If the *PRT_UNIT* argument is absent, then all output is on the unit `defunit` defined in the *Select_Parameters* module.

If the argument *TRACE* is `true`, it also outputs a message that the routine identified by *STRING* was entered.

Synopsis:

```
message = entering( string , level , trace , prt_unit=prt_unit )
```

leaving()

purpose:

leaving() is the *opposite* to `entering`. It should be called just before leaving a routine. The argument *LEVEL* is reduced by `2` and if the argument *TRACE* is `true`, an exit message is output.

Trailing blanks in *STRING* are removed. If the *PRT_UNIT* argument is absent, then all output is on the unit `defunit` defined in the *Select_Parameters* module.

Synopsis:

```
call leaving( string , level , trace , prt_unit=prt_unit )
```

indent()

purpose:

indent() can also used to indent output, albeit in a manner different from `entering` and `leaving`.

It simply writes out *LEVEL* blanks followed by the string *ID* in `[]`, and leaves the output file marker where it is. It uses nonadvancing output.

If the *LEVEL* argument is not present, just the *ID* part is output; i.e. *LEVEL* is treated as zero.

Leading and trailing blanks in *ID* are removed.

If the *PRT_UNIT* argument is absent, then all output is on the unit `defunit` defined in the *Select_Parameters* module.

Synopsis:

```
call indent(id , level=level , prt_unit=prt_unit )
```

write_array()

purpose:

write_array() prints out an integer or real array *X* (e.g. a vector or matrix) with a given format and a title, as given in its input arguments.

The integer or real array *X* is printed row by row.

If the *PRT_UNIT* argument is absent, then all output is on the unit `defunit` defined in the *Select_Parameters* module.

Synopsis:

```
call write_array( x(:),   f=f , w=w , d=d , s=s , name=name , indent=indent ,␣
↪line=line , prt_unit=prt_unit )   ! x is a real array
call write_array( x(:,:), f=f , w=w , d=d , s=s , name=name , indent=indent ,␣
↪line=line , prt_unit=prt_unit )   ! x is a real array
call write_array( x(:),         w=w ,       s=s , name=name , indent=indent ,␣
↪line=line , prt_unit=prt_unit )   ! x is an integer array
call write_array( x(:,:),       w=w ,       s=s , name=name , indent=indent ,␣
↪line=line , prt_unit=prt_unit )   ! x is an integer array
```

print_array()

purpose:

print_array() is a routine for labeled integer or real matrix/vector output, with given format and title, as given in its input arguments.

The integer or real array is printed columns block by columns block.

If the *PRT_UNIT* argument is absent, then all output is on the unit `defunit` defined in the *Select_Parameters* module.

Synopsis:

```
call print_array( x(:p),     f=f , w=w , d=d , sign_ed=sign_ed ,       ␣
↪title=title , namlig=namlig(:p) ,                    indent=indent ,    ␣
↪         prt_unit=prt_unit )   ! x is a real array
call print_array( x(:p,:n) , f=f , w=w , d=d , sign_ed=sign_ed , s=s ,␣
↪title=title , namlig=namlig(:p) , namcol=namcol(:n) , indent=indent ,␣
↪line=line , prt_unit=prt_unit )   ! x is a real array
call print_array( x(:),           w=w ,       sign_ed=sign_ed ,       ␣
↪title=title , namlig=namlig(:p) ,                    indent=indent ,    ␣
↪         prt_unit=prt_unit )   ! x is an integer array
call print_array( x(:p,:n) ,      w=w ,       sign_ed=sign_ed , s=s ,␣
↪title=title , namlig=namlig(:p) , namcol=namcol(:n) , indent=indent ,␣
↪line=line , prt_unit=prt_unit )   ! x is an integer array
```

Examples:

ex1_print_array.F90

print_prinfac()

purpose:

print_prinfac() is a routine for labeled matrix output after an EOF or SVD analysis.

Print an EOF model (*MODE* = 1) or the associated principal components (*MODE* = 2) and an SVD model (*MODE* = 3) or the associated singular variables (*MODE* = 4).

If the *PRT_UNIT* argument is absent, then all output is on the unit `defunit` defined in the *Select_Parameters* module.

Synopsis:

```
call print_prinfac( mode, a(:p) ,    f=f , names=names(:p) ,             prt_
↪unit=prt_unit )
call print_prinfac( mode, a(:p,:n) , f=f , names=names(:p) , line=line , prt_
↪unit=prt_unit )
```

print_stat()

purpose:

print_stat() prints statistics for an EOF "missing" or "weighted" analysis, for

- Variables (*MODE* = 1)
- Observations (*MODE* = 2)

If the *PRT_UNIT* argument is absent, then all output is on the unit `defunit` defined in the *Select_Parameters* module.

Synopsis:

```
call print_stat( mode , nomiss(:p) , var(:p) , inr(:p) , qlt(:p) , names(:p) ,
↪ prt_unit=prt_unit )
call print_stat ( mode , weight(:p) , var(:p) , inr(:p) , qlt(:p) , names(:p)␣
↪, prt_unit=prt_unit )
```

5.11 MODULE String_Procedures

Module *String_Procedures* exports constants, subroutines and functions for manipulating strings.

Here is the list of the public constants exported by module *String_Procedures*:

```
!
! CODES FOR CASE CONVERSIONS.
!
integer(i1b), parameter ::      &
    toupper       = 1,          &
    tolower       = 2,          &
    capitalize    = 3
!
! CODES FOR NUMERICAL DATA TYPES STORED IN A STRING.
!
integer(i1b), parameter  ::     &
    kchr = 0,                   & ! NON-NUMERICAL STRING
    kint = 1,                   & ! INTEGER
    kfix = 2,                   & ! FIXED REAL
    kexp = 3                      ! REAL WITH EXPONENT
```

In order to use one of these constants or one of the routines listed below, you must include an appropriate `use String_Procedures` or `use Statpack` statement in your Fortran program, like:

```
use String_Procedures, only: capitalize
```

or :

```
use Statpack, only: capitalize
```

Here is the list of the public routines exported by module *String_Procedures*:

ascii_is_upper()

Synopsis:

```
c_is_upper = ascii_is_upper( c )
```

is_upper()

Synopsis:

```
c_is_upper = is_upper( c )
```

ascii_is_lower()

Synopsis:

```
c_is_lower = ascii_is_lower( c )
```

is_lower()

Synopsis:

```
c_is_lower = is_lower( c )
```

ascii_is_alpha()

Synopsis:

```
c_is_alpha = ascii_is_alpha( c )
```

is_alpha()

Synopsis:

```
c_is_alpha = is_alpha( c )
```

ascii_is_same()

Synopsis:

```
c1_c2_are_same = ascii_is_same( c1 , c2 )
```

is_same()

Synopsis:

```
c1_c2_are_same = is_same( c1 , c2 )
```

ascii_is_digit()

Synopsis:

```
c_is_digit = ascii_is_digit( c )
```

is_digit()

Synopsis:

```
c_is_digit = is_digit( c )
```

is_space()

Synopsis:

```
c_is_space = is_space( c )
```

is_num()

Synopsis:

```
string_is_num = is_num( string )
```

string_count()

Synopsis:

```
count = string_count( string , letter )
```

ascii_string_eq()

Synopsis:

```
strings_are_same = ascii_string_eq( string1 , string2 )
```

string_eq()

Synopsis:

```
strings_are_same = string_eq( string1 , string2 )
```

ascii_string_index()

Synopsis:

```
index = ascii_string_index( string , list(:) )
```

string_index()

Synopsis:

```
index = string_index( string , list(:) )
```

ascii_string_comp()

Synopsis:

```
compare_strings = ascii_string_comp( string1 , string2 )
```

string_comp()

Synopsis:

```
compare_strings = string_comp( string1 , string2 )
```

ebc2asc()

Synopsis:

```
call ebc2asc( ebc_str , asc_str , nchr )
```

asc2ebc()

Synopsis:

```
call asc2ebc( asc_str , ebc_str , nchr )
```

ascii_to_upper()

Synopsis:

```
c_upper = ascii_to_upper( c )
```

to_upper()

Synopsis:

```
c_upper = to_upper( c )
```

ascii_to_lower()

Synopsis:

```
c_lower = ascii_to_lower( c )
```

to_lower()

Synopsis:

```
c_lower = to_lower( c )
```

ascii_case_change()

Synopsis:

```
call ascii_case_change( string , type )
```

case_change()

Synopsis:

```
call case_change( string , type )
```

mid_shift()

Synopsis:

```
call mid_shift( string , from , to , number )
```

center()

Synopsis:

```
call center( string )
```

find_field()

Synopsis:

```
call find_field( string , istart , iend , delims=delims , isearch=isearch )
```

nbrchf()

Synopsis:

```
nchar = nbrchf( jval )
nchar = nbrchf( rval )
```

obt_fmt()

Synopsis:

```
fmt = obt_fmt( jval )
fmt = obt_fmt( rval )
```

val_to_string()

Synopsis:

```
call val_to_string( jval , string , nchar                  )
call val_to_string( rval , string , nchar, fmt=fmt , d=d  )
```

string_to_val()

Synopsis:

```
call string_to_val( string , kcode , fmt )
```

5.12 MODULE Time_Procedures

Module *Time_Procedures* exports constants, subroutines and functions for manipulating dates and time.

Here is the list of the public constants exported by module *Time_Procedures*:

```
!
! MONTHS OF THE YEAR.
!
character(len=9), parameter :: months(12) = (/   'January  ', 'February ',  &
                                                 'March    ', 'April    ',  &
                                                 'May      ', 'June     ',  &
                                                 'July     ', 'August   ',  &
                                                 'September', 'October  ',  &
                                                 'November ', 'December '/)
!
! DAYS OF THE WEEK.
!
character(len=9), parameter :: days(1:7)  = (/   'Monday   ',               &
                                                 'Tuesday  ', 'Wednesday',  &
                                                 'Thursday ', 'Friday   ',  &
                                                 'Saturday ', 'Sunday   ' /)
```

In order to use one of these constants or one of the routines listed below, you must include an appropriate `use Time_Procedures` or `use Statpack` statement in your Fortran program, like:

```
use Time_Procedures, only: months
```

or :

```
use Statpack, only: months
```

Here is the list of the public routines exported by *Time_Procedures*:

leapyr()

Purpose:

leapyr() checks for a leap year. *LEAPYR* is returned as `true` if *IYR* is a leap year, and `false` otherwise.

This function uses the Gregorian calendar adopted the Oct. 15, 1582.

Leap years are years that are evenly divisible by 4, except years that are evenly divisible by 100 must be divisible by 400.

Synopsis:

```
is_leap_year = leapyr( iyr )
```

Examples:

ex1_leapyr.F90

daynum()

Purpose:

daynum() computes a day number.

One of the more useful applications for this routine is to compute the number of days between two dates.

This function uses the Gregorian calendar adopted the Oct. 15, 1582.

In other words, Oct. 15, 1582 will return a day number of unity and hence this algorithm will not work properly for dates early than 10-15-1582.

Synopsis:

```
jdaynum = daynum( iyr , imon , iday )
```

day_of_week()

Purpose:

day_of_week() returns the day of the week (e.g., Mon, Tue, . . .) as an index (e.g. Mon = 1 to Sun=7) for a given year, month, and day.

This routine assumes a valid day, month and year are input.

Synopsis:

```
wdaynum = day_of_week( iyr , imon , iday )
```

Examples:

ex1_day_of_week.F90

daynum_to_ymd()

Purpose:

daynum_to_ymd() converts a Julian Day Number (*JDAYNUM*) to Gregorian year (*IYR*), month (*IMON*) and day (*IDAY*) in the Gregorian calendar promulgated by Gregory XIII, starting with `jdaynum = 1` on Friday, 15 October 1582.

This subroutine is adapted from a MATLAB M-file written by W. Kahan available on the WEB.

Synopsis:

```
call daynum_to_ymd( jdaynum , iyr , imon , iday )
```

Examples:

ex1_daynum_to_ymd.F90

ymd_to_daynum()

Purpose:

ymd_to_daynum() is just the opposite of *daynum_to_ymd()*. It converts Gregorian year (*IYR*), month (*IMON*) and day (*IDAY*) to Julian day Number.

ymd_to_daynum() is useful to compute the number of days between two dates, which is the difference between their Julian day.

This subroutine is adapted from a MATLAB M-file written by W. Kahan available on the WEB.

Synopsis:

```
jdaynum = ymd_to_daynum( iyr , imon , iday )
```

Examples:

ex1_ymd_to_daynum.F90

ymd_to_dayweek()

Purpose:

ymd_to_dayweek() computes the day of the week from Gregorian year (*IYR*), month (*IMON*) and day (*IDAY*), as an integer index (e.g. Mon=1 to Sun=7) for the given year, month, and day in the Gregorian calendar promulgated by Gregory XIII on Friday, 15 October 1582.

This subroutine is adapted from a MATLAB M-file written by W. Kahan available on the WEB.

Synopsis:

```
wdaynum = ymd_to_dayweek( iyr , imon , iday )
```

Examples:

ex1_ymd_to_dayweek.F90

daynum_to_dayweek()

Purpose:

daynum_to_dayweek() computes the day of the week from Julian day number *JDAYNUM*, as an integer index (e.g. Mon=1 to Sun=7) starting with `jdaynum = 1` on Friday, 15 October 1582.

Synopsis:

```
wdaynum = daynum_to_dayweek( jdaynum )
```

Examples:

ex1_daynum_to_dayweek.F90

rtsw()

Purpose:

rtsw() is a Real-Time Stop Watch.

This routine can be used to compute the time lapse (in seconds) between functions calls according to the system (wall) clock.

Since this routine uses the system clock, the elapsed time computed with this routine may not (probably won't be in a multi-tasking OS) an accurate reflection of the number of cpu cycles required to perform a calculation. Therefore care should be exercised when using this to profile a code.

The result is a real of kind **extd**.

Synopsis:

```
wtime = rtsw( )
```

Examples:

ex1_rtsw.F90

elapsed_time()

Purpose:

elapsed_time() computes elapsed time between two invocations of the intrinsic function **date_and_time()**. `elapsed_time( t1, t0 )` returns the time in seconds that has elapsed between the vectors *T0* and *T1*. Each vector must have at least seven elements in the format returned by **date_and_time()** for the optional argument *VALUES*; namely

```
T = (/ year, month, day, x, hour, minute, second /)
```

This routine can be used to compute the elapsed time between **date_and_time()** calls according to the system (wall) clock.

Since this routine uses the system clock, the elapsed time computed with this routine may not (probably won't be in a multi-tasking OS) an accurate reflection of the number of cpu cycles required to perform a calculation. Therefore care should be exercised when using this to profile a code.

Synopsis:

```
etime = elapsed_time( t1(:n) , t0(:n) )
```

Examples:

ex1_elapsed_time.F90

cpusecs()

Purpose:

cpusecs() obtains, from the intrinsic routine **system_clock()**, the current value of the system CPU usage clock. This value is then converted to seconds and returned as an extended precision real value (e.g. of kind **extd**).

This functions assumes that the number of CPU cycles (e.g. clock counts) between two calls is less than *COUNT_MAX*, the maximum possible value of clock counts as returned by the intrinsic routine **system_clock()**.

The result is a real of kind **extd**.

Synopsis:

```
cputime = cpusecs( )
```

Examples:

ex1_cpusecs.F90

time_to_hmsms()

Purpose:

time_to_hmsms() converts time (in seconds) to hours, minutes, seconds, milliseconds format.

Synopsis:

```
call time_to_hmsms( time , hmsms(:4) )
```

time_to_string()

Purpose:

time_to_string() converts *TIME* to a string format for printing as

```
    hours.minutes.seconds.milliseconds
```

The result is a string of (at least) 13 characters.

Synopsis:

```
ctime = time_to_string( time )
```

Examples:

ex1_time_to_string.F90

get_date()

Purpose:

get_date() outputs a given date given as year (*IYR*), month (*IMON*), and day (*IDAY*) in a *nice* format.

Synopsis:

```
call get_date( iyr , imon , iday , date )
```

get_date_time()

Purpose:

get_date_time() outputs system date and time in *nice* formats.

This routine just reformats the output from the standard **date_and_time()** intrinsic function.

Synopsis:

```
call get_date_time( date=date, time=time )
```

system_date_time()

Purpose:

system_date_time() retrieves the current system time and date and transfers them to the string argument *CHDATE* in a "pretty" format, i.e.,

```
chdate = "DATE: DD-MMM-YYYY TIME: HH:MM:SS"
```

If the *CHDATE* argument is more than 33 characters in length, *CHDATE* is padded with blanks. If it is less than 33 in length, only the leftmost characters of the date will be returned.

Synopsis:

```
call system_date_time( chdate )
```

my_date_time()

Purpose:

my_date_time() returns in *CHDATE* a 41-character date of the form given in model (below).

It uses the time and date as obtained from the intrinsic routine **date_and_time()** and converts them to the form of the model:

```
chdate = "00:00 a.m., Wednesday, September 00, 1999"
```

Note that excess blanks in the date are eliminated. If *CHDATE* is more than 41 characters in length, *CHDATE* is padded with blanks. If it is less than 41 in length, only the leftmost characters of the date will be returned.

Synopsis:

```
call my_date_time( chdate )
```

5.13 MODULE Utilities

Module *Utilities* Utilities exports subroutines and functions for simple and general computations.

Some of these routines can be used in place of intrinsic functions in the STATPACK library, like the *dot_product2()*, *transpose2()* and *matmul2()* functions described below, if the cpp macros _DOT_PRODUCT, _TRANSPOSE and _MATMUL are activated at compilation of the STATPACK library. See the section *Preprocessor cpp macros* for more details.

A large set of accurate routines for computing the 2-norm (i.e., the Euclidean norm) of (real or complex) vectors or the Frobenius norm of (real or complex) matrices using up-to-date algorithms with due regard to avoiding overflow and underflow [Anderson:2002] [Anderson:2018] [Hanson_Hopkins:2018] are also included, like the *norm()*, *norme()*, *norm2e()*, *lassq()*, *lassqe()*, *lassq2e()* routines described below.

Some of these routines are also adapted from public domain routines in Numerical Recipes.

Note, finally, that many of these routines are low-level routines, which do not include checking of the correctness of the size/shape of their array arguments for enhanced speed at execution. This means that the user must exercise care when using these low-level subroutines and functions.

In order to use one of these routines, you must include an appropriate `use Utilities` or `use Statpack` statement in your Fortran program, like:

```
use Utilities, only: transpose2
```

or :

```
use Statpack, only: transpose2
```

In order to replace the calls to the intrinsic functions **dot_product**(), **transpose**() or **matmul**() by the corresponding STATPACK functions *dot_product2()*, *transpose2()* and *matmul2()* in your Fortran program, include in your program a statement like:

```
use Utilities, only: transpose=>transpose2
```

or :

```
use Statpack, only: transpose=>transpose2
```

Here is the list of the public routines exported by module *Utilities*:

transpose2()

purpose:

transpose2() computes MAT^T for a given input (real, complex, integer or logical) matrix MAT.

If the cpp macro _OPENMP is activated at compilation, **transpose2**() will be parallelized with OpenMP if the input matrix is big enough.

Synopsis:

```
mat_t(:m,:n) = transpose2( mat(:n,:m) )  ! mat is a  real    matrix of kind␣
↪stnd
mat_t(:m,:n) = transpose2( mat(:n,:m) )  ! mat is a  complex matrix of kind␣
↪stnd
mat_t(:m,:n) = transpose2( mat(:n,:m) )  ! mat is an integer matrix of kind␣
↪i4b
mat_t(:m,:n) = transpose2( mat(:n,:m) )  ! mat is a  logical matrix of kind␣
↪lgl
```

Examples:

ex1_transpose2.F90

dot_product2()

purpose:

dot_product2() computes the scalar product of two input (real, complex, integer or logical) vectors.

dot_product2() will use BLAS1 subroutines through the *BLAS_interfaces* module for computing the dot product for real or complex arguments if the cpp macro _BLAS is activated at compilation of the STATPACK library.

Synopsis:

```
xy = dot_product2( vecx(:n) , vecy(:n) )  ! vecx and vecy are real    vectors␣
↪of kind stnd
xy = dot_product2( vecx(:n) , vecy(:n) )  ! vecx and vecy are complex vectors␣
↪of kind stnd
xy = dot_product2( vecx(:n) , vecy(:n) )  ! vecx and vecy are integer vectors␣
↪of kind i4b
xy = dot_product2( vecx(:n) , vecy(:n) )  ! vecx and vecy are logical vectors␣
↪of kind lgl
```

mmproduct()

purpose:

mmproduct() multiplies the two input (real, complex, integer or logical) matrices or vectors.

Synopsis:

```
array(:m)    = mmproduct( vec(:n)     , mat(:n,:m)  ) ! vec and mat   are␣
↪real    arrays of kind stnd
array(:n)    = mmproduct( mat(:n,:m)  , vec2(:m)    ) ! mat and vec2  are␣
↪real    arrays of kind stnd
array(:n,:m) = mmproduct( mat1(:n,:p) , mat2(:p,:m) ) ! mat1 and mat2 are␣
↪real    arrays of kind stnd
array(:m)    = mmproduct( vec(:n)     , mat(:n,:m)  ) ! vec and mat   are␣
↪complex arrays of kind stnd
array(:n)    = mmproduct( mat(:n,:m)  , vec2(:m)    ) ! mat and vec2  are␣
↪complex arrays of kind stnd
array(:n,:m) = mmproduct( mat1(:n,:p) , mat2(:p,:m) ) ! mat1 and mat2 are␣
↪complex arrays of kind stnd
```

matmul2()

purpose:

matmul2() multiplies the two input (real, complex, integer or logical) matrices or vectors.

matmul2() will use BLAS2 or BLAS3 subroutines through the *BLAS_interfaces* module for performing the multiplication for real or complex arguments if the cpp macro _BLAS is activated at compilation of the STATPACK library.

On the other hand, if the cpp macros _BLAS and _NOOPENMP3 are not activated, but the cpp macro _OPENMP is activated at compilation, **matmul2()** will be parallelized with OpenMP if the input matrices or vectors are big enough.

Synopsis:

```
array(:m)    = matmul2( vec(:n)     , mat(:n,:m)  ) ! vec and mat   are real ␣
↪  arrays of kind stnd
array(:n)    = matmul2( mat(:n,:m)  , vec2(:m)    ) ! mat and vec2  are real ␣
↪  arrays of kind stnd
array(:n,:m) = matmul2( mat1(:n,:p) , mat2(:p,:m) ) ! mat1 and mat2 are real ␣
↪  arrays of kind stnd
array(:m)    = matmul2( vec(:n)     , mat(:n,:m)  ) ! vec and mat   are␣
↪complex arrays of kind stnd
array(:n)    = matmul2( mat(:n,:m)  , vec2(:m)    ) ! mat and vec2  are␣
↪complex arrays of kind stnd
array(:n,:m) = matmul2( mat1(:n,:p) , mat2(:p,:m) ) ! mat1 and mat2 are␣
↪complex arrays of kind stnd
array(:m)    = matmul2( vec(:n)     , mat(:n,:m)  ) ! vec and mat   are real ␣
↪  arrays of kind i4b
array(:n)    = matmul2( mat(:n,:m)  , vec2(:m)    ) ! mat and vec2  are real ␣
↪  arrays of kind i4b
array(:n,:m) = matmul2( mat1(:n,:p) , mat2(:p,:m) ) ! mat1 and mat2 are real ␣
↪  arrays of kind i4b
array(:m)    = matmul2( vec(:n)     , mat(:n,:m)  ) ! vec and mat   are␣
↪logical arrays of kind lgl
array(:n)    = matmul2( mat(:n,:m)  , vec2(:m)    ) ! mat and vec2  are␣
↪logical arrays of kind lgl
array(:n,:m) = matmul2( mat1(:n,:p) , mat2(:p,:m) ) ! mat1 and mat2 are␣
↪logical arrays of kind lgl
```

Examples:

ex1_matmul2.F90

array_copy()

purpose:

array_copy() makes a (truncated) copy of the input one-dimensional array *SRC* into the output one-dimensional array *DEST*

Synopsis:

```
call array_copy( src(:) , dest(:) , n_copied, n_not_copied ) ! src and dest␣
↪are integer vectors of kind i4b
call array_copy( src(:) , dest(:) , n_copied, n_not_copied ) ! src and dest␣
↪are real    vectors of kind stnd
call array_copy( src(:) , dest(:) , n_copied, n_not_copied ) ! src and dest␣
↪are complex vectors of kind stnd
```

swap()

purpose:

swap() swaps the corresponding elements of the two input arguments *A* and *B*.

For real or complex array arguments, **swap()** will use BLAS1 subroutines through the *BLAS_interfaces* module when possible if the cpp macro _BLAS is activated at compilation of the STATPACK library.

Synopsis:

```
call swap( a        , b                    ) ! a and b are integers of kind␣
↪i4b
call swap( a        , b                    ) ! a and b are reals    of kind␣
↪stnd
call swap( a        , b                    ) ! a and b are complex  of kind␣
↪stnd
call swap( a(:n)    , b(:n)                ) ! a and b are integer vectors␣
↪of kind i4b
call swap( a(:n)    , b(:n)                ) ! a and b are real vectors   ␣
↪of kind stnd
call swap( a(:n)    , b(:n)                ) ! a and b are complex vectors␣
↪of kind stnd
call swap( a(:n,:m) , b(:n,:m)             ) ! a and b are integer matrices␣
↪of kind i4b
call swap( a(:n,:m) , b(:n,:m)             ) ! a and b are real matrices   ␣
↪of kind stnd
call swap( a(:n,:m) , b(:n,:m)             ) ! a and b are complex matrices␣
↪of kind stnd
call swap( a        , b        , mask      ) ! a and b are integers of kind␣
↪i4b
call swap( a        , b        , mask      ) ! a and b are reals    of kind␣
↪stnd
call swap( a        , b        , mask      ) ! a and b are complex  of kind␣
↪stnd
call swap( a(:n)    , b(:n)    , mask(:n)  ) ! a and b are integer vectors␣
↪of kind i4b
call swap( a(:n)    , b(:n)    , mask(:n)  ) ! a and b are real vectors   ␣
↪of kind stnd
call swap( a(:n)    , b(:n)    , mask(:n)  ) ! a and b are complex vectors␣
↪of kind stnd
```

```
call swap( a(:n,:m) , b(:n,:m) , mask(:n,:m) ) ! a and b are integer matrices␣
↪of kind i4b
call swap( a(:n,:m) , b(:n,:m) , mask(:n,:m) ) ! a and b are real matrices   ␣
↪of kind stnd
call swap( a(:n,:m) , b(:n,:m) , mask(:n,:m) ) ! a and b are complex matrices␣
↪of kind stnd
```

mvalloc()

purpose:

mvalloc() reallocates an allocatable vector or matrix with a new size or shape, while preserving its contents. **mvalloc()** is only available if the cpp macro _F2003 is activated at compilation of the STATPACK library.

Synopsis:

```
call mvalloc( p(:)   , n     , ialloc )  ! p is an allocated integer vector␣
↪of kind i4b
call mvalloc( p(:)   , n     , ialloc )  ! p is an allocated real vector of␣
↪kind stnd
call mvalloc( p(:)   , n     , ialloc )  ! p is an allocated complex vector␣
↪of kind stnd
call mvalloc( p(:)   , n     , ialloc )  ! p is an allocated character vector
call mvalloc( p(:,:) , n , m , ialloc )  ! p is an allocated integer matrix␣
↪of kind i4b
call mvalloc( p(:,:) , n , m , ialloc )  ! p is an allocated real matrix of␣
↪kind stnd
call mvalloc( p(:,:) , n , m , ialloc )  ! p is an allocated complex matrix␣
↪of kind stnd
```

ifirstloc()

Synopsis:

```
index = ifirstloc( mask(:) )
```

imaxloc()

Synopsis:

```
index = imaxloc( arr(:n)           ) ! arr is an integer array of kind i4b
index = imaxloc( arr(:n) , mask(:n) ) ! arr is an integer array of kind i4b
index = imaxloc( arr(:n)           ) ! arr is a   real array   of kind stnd
index = imaxloc( arr(:n) , mask(:n) ) ! arr is a   real array   of kind stnd
```

iminloc()

Synopsis:

```
index = iminloc( arr(:n)           ) ! arr is an integer array of kind i4b
index = iminloc( arr(:n) , mask(:n) ) ! arr is an integer array of kind i4b
index = iminloc( arr(:n)           ) ! arr is a   real array   of kind stnd
index = iminloc( arr(:n) , mask(:n) ) ! arr is a   real array   of kind stnd
```

assert()

Synopsis:

```
call assert( n1 ,                string )
call assert( n1 , n2 ,           string )
call assert( n1 , n2 , n3 ,      string )
call assert( n1 , n2 , n3 , n4 , string )
```

```
call assert( n(:) ,              string )
```

assert_eq()

Synopsis:

```
n = assert_eq( n1 , n2 ,           string )
n = assert_eq( n1 , n2 , n3 ,      string )
n = assert_eq( n1 , n2 , n3 , n4 , string )
n = assert_eq( nn(:) ,             string )
```

merror()

Synopsis:

```
call merror( string , ierror=ierror )
```

arth()

Synopsis:

```
vec(:n)    = arth( first     , increment     , n ) ! first and increment are␣
↪integers of kind i4b
vec(:n)    = arth( first     , increment     , n ) ! first and increment are␣
↪reals    of kind stnd
vec(:n)    = arth( first     , increment     , n ) ! first and increment are␣
↪complex  of kind stnd
vec(:m,:n) = arth( first(:m) , increment(:m) , n ) ! first and increment are␣
↪integer vectors of kind i4b
vec(:m,:n) = arth( first(:m) , increment(:m) , n ) ! first and increment are␣
↪real vectors    of kind stnd
vec(:m,:n) = arth( first(:m) , increment(:m) , n ) ! first and increment are␣
↪complex vectors of kind stnd
```

geop()

Synopsis:

```
vec(:n)    = geop( first     , factor     , n ) ! first and factor are␣
↪integers of kind i4b
vec(:n)    = geop( first     , factor     , n ) ! first and factor are reals ␣
↪  of kind stnd
vec(:n)    = geop( first     , factor     , n ) ! first and factor are␣
↪complex  of kind stnd
vec(:m,:n) = geop( first(:m) , factor(:m) , n ) ! first and factor are␣
↪integer vectors of kind i4b
vec(:m,:n) = geop( first(:m) , factor(:m) , n ) ! first and factor are real␣
↪vectors    of kind stnd
vec(:m,:n) = geop( first(:m) , factor(:m) , n ) ! first and factor are␣
↪complex vectors of kind stnd
```

cumsum()

Synopsis:

```
vec(:n) = cumsum( arr(:n) , seed ) ! arr is an integer array of kind i4b
vec(:n) = cumsum( arr(:n) , seed ) ! arr is a  real    array of kind stnd
vec(:n) = cumsum( arr(:n) , seed ) ! arr is a  complex array of kind stnd
```

cumprod()

Synopsis:

```
vec(:n) = cumprod( arr(:n) , seed ) ! arr is an integer array of kind i4b
vec(:n) = cumprod( arr(:n) , seed ) ! arr is a  real    array of kind stnd
vec(:n) = cumprod( arr(:n) , seed ) ! arr is a  complex array of kind stnd
```

poly()

Synopsis:

```
y     = poly( x     , coeffs(:)            ) ! x is a real    scalar of kind␣
↪stnd and coeffs is a real    array of kind stnd
y     = poly( x     , coeffs(:)            ) ! x is a complex scalar of kind␣
↪stnd and coeffs is a real    array of kind stnd
y     = poly( x     , coeffs(:)            ) ! x is a complex scalar of kind␣
↪stnd and coeffs is a complex array of kind stnd
y(:n) = poly( x(:n) , coeffs(:)            ) ! x and coeffs are real arrays␣
↪of kind stnd
y(:n) = poly( x(:n) , coeffs(:) , mask(:n) ) ! x and coeffs are real arrays␣
↪of kind stnd and mask is a logical array of kind lgl
```

poly_term()

Synopsis:

```
y(:n) = poly_term( coeffs(:n) , x ) ! x is a real    scalar of kind stnd and␣
↪coeffs is a real    array of kind stnd
y(:n) = poly_term( coeffs(:n) , x ) ! x is a complex scalar of kind stnd and␣
↪coeffs is a complex array of kind stnd
```

zroots_unity()

Synopsis:

```
x(:nn) = zroots_unity( n, nn )
```

update_rk1()

Synopsis:

```
call update_rk1( mat(:m,:n) , u(:m) , v(:n) ) ! all are integer arrays of␣
↪kind i4b
call update_rk1( mat(:m,:n) , u(:m) , v(:n) ) ! all are real    arrays of␣
↪kind stnd
call update_rk1( mat(:m,:n) , u(:m) , v(:n) ) ! all are complex arrays of␣
↪kind stnd
```

update_rk2()

Synopsis:

```
call update_rk2( mat(:m,:n) , u(:m) , v(:n) , u2(:m) , v2(:n) ) ! all are␣
↪integer arrays of kind i4b
call update_rk2( mat(:m,:n) , u(:m) , v(:n) , u2(:m) , v2(:n) ) ! all are␣
↪real    arrays of kind stnd
call update_rk2( mat(:m,:n) , u(:m) , v(:n) , u2(:m) , v2(:n) ) ! all are␣
↪complex arrays of kind stnd
```

outerprod()

Synopsis:

```
mat(:n,:m) =  outerprod( a(:n) , b(:m) ) ! a and b are integer arrays of kind␣
↪i4b
```

```
mat(:n,:m) =  outerprod( a(:n) , b(:m) ) ! a and b are real    arrays of kind␣
↪stnd
mat(:n,:m) =  outerprod( a(:n) , b(:m) ) ! a and b are complex arrays of kind␣
↪stnd
```

outerdiv()

Synopsis:

```
mat(:n,:m) =  outerdiv( a(:n) , b(:m) ) ! a and b are real    arrays of kind␣
↪stnd
mat(:n,:m) =  outerdiv( a(:n) , b(:m) ) ! a and b are complex arrays of kind␣
↪stnd
```

outersum()

Synopsis:

```
mat(:n,:m) =  outersum( a(:n) , b(:m) ) ! a and b are integer arrays of kind␣
↪i4b
mat(:n,:m) =  outersum( a(:n) , b(:m) ) ! a and b are real    arrays of kind␣
↪stnd
mat(:n,:m) =  outersum( a(:n) , b(:m) ) ! a and b are complex arrays of kind␣
↪stnd
```

outerdiff()

Synopsis:

```
mat(:n,:m) = outerdiff( a(:n) , b(:m) ) ! a and b are integer arrays of kind␣
↪i4b
mat(:n,:m) = outerdiff( a(:n) , b(:m) ) ! a and b are real    arrays of kind␣
↪stnd
mat(:n,:m) = outerdiff( a(:n) , b(:m) ) ! a and b are complex arrays of kind␣
↪stnd
```

outerand()

Synopsis:

```
mat(:n,:m) = outerand( a(:n) , b(:m) ) ! a and b are logical arrays of kind␣
↪lgl
```

outeror()

Synopsis:

```
mat(:n,:m) = outeror( a(:n) , b(:m) ) ! a and b are logical arrays of kind lgl
```

triangle()

Synopsis:

```
mat(:j,:k) = triangle( upper , j , k , extra=extra )
```

Examples:

ex2_trid_inviter.F90

ex2_trid_deflate.F90

abse()

purpose:

abse() computes the 2-norm (i.e., the Euclidean norm) of a (real or complex) vector or the Frobenius norm of a (real or complex) matrix.

abse() is based on methods from [Hanson_Hopkins:2018] and uses compensated summation in order to minimize rounding errors.

Synopsis:

```
l2norm     = abse( vec(:)   ) ! vec is real    array of kind stnd
l2norm     = abse( vec(:)   ) ! vec is complex array of kind stnd
fnorm      = abse( mat(:,:) ) ! mat is real    array of kind stnd
fnorm      = abse( mat(:,:) ) ! mat is complex array of kind stnd
l2norm(:)  = abse( mat(:,:) , dim ) ! mat is real    array of kind stnd
l2norm(:)  = abse( mat(:,:) , dim ) ! mat is complex array of kind stnd
```

norm()

purpose:

norm() computes the 2-norm (i.e., the Euclidean norm) of a (real or complex) vector or the Frobenius norm of a (real or complex) matrix.

norm() is based on an updated version of the Blue's algorithm [Blue:1978] [Anderson:2018].

Synopsis:

```
l2norm     = norm( vec(:)   ) ! vec is real    array of kind stnd
l2norm     = norm( vec(:)   ) ! vec is complex array of kind stnd
fnorm      = norm( mat(:,:) ) ! mat is real    array of kind stnd
fnorm      = norm( mat(:,:) ) ! mat is complex array of kind stnd
l2norm(:)  = norm( mat(:,:) , dim ) ! mat is real    array of kind stnd
l2norm(:)  = norm( mat(:,:) , dim ) ! mat is complex array of kind stnd
```

lassq()

purpose:

lassq() computes a scaled sum of squares based on an updated version of the Blue's algorithm [Blue:1978] [Anderson:2018].

Synopsis:

```
call lassq( vec(:)   , scal, ssq ) ! vec is real    array of kind stnd
call lassq( vec(:)   , scal, ssq ) ! vec is complex array of kind stnd
call lassq( mat(:,:) , scal, ssq ) ! mat is real    array of kind stnd
call lassq( mat(:,:) , scal, ssq ) ! mat is complex array of kind stnd
```

norme()

purpose:

norme() computes the 2-norm (i.e., the Euclidean norm) of a (real or complex) vector or the Frobenius norm of a (real or complex) matrix.

norme() is based on an updated version of the Blue's algorithm [Blue:1978] [Anderson:2018] and uses compensated summation in order to minimize rounding errors [Hanson_Hopkins:2018].

Synopsis:

```
l2norm     = norme( vec(:)   ) ! vec is real    array of kind stnd
l2norm     = norme( vec(:)   ) ! vec is complex array of kind stnd
fnorm      = norme( mat(:,:) ) ! mat is real    array of kind stnd
fnorm      = norme( mat(:,:) ) ! mat is complex array of kind stnd
l2norm(:)  = norme( mat(:,:) , dim ) ! mat is real    array of kind stnd
```

```
l2norm(:)  = norme( mat(:,:) , dim ) ! mat is complex array of kind stnd
```

lassqe()

purpose:

lassqe() computes a scaled sum of squares based on an updated version of the Blue's algorithm [Blue:1978] [Anderson:2018] and uses compensated summation in order to minimize rounding errors [Hanson_Hopkins:2018].

Synopsis:

```
call lassqe( vec(:)   , scal, ssq ) ! vec is real    array of kind stnd
call lassqe( vec(:)   , scal, ssq ) ! vec is complex array of kind stnd
call lassqe( mat(:,:) , scal, ssq ) ! mat is real    array of kind stnd
call lassqe( mat(:,:) , scal, ssq ) ! mat is complex array of kind stnd
```

norm2e()

purpose:

norm2e() computes the 2-norm (i.e., the Euclidean norm) of a (real or complex) vector or the Frobenius norm of a (real or complex) matrix.

norm2e() is based on an updated version of the LAPACK3E' algorithm [Anderson:2002] [Anderson:2018] and uses compensated summation in order to minimize rounding errors [Hanson_Hopkins:2018].

Synopsis:

```
l2norm     = norm2e( vec(:)   ) ! vec is real    array of kind stnd
l2norm     = norm2e( vec(:)   ) ! vec is complex array of kind stnd
fnorm      = norm2e( mat(:,:) ) ! mat is real    array of kind stnd
fnorm      = norm2e( mat(:,:) ) ! mat is complex array of kind stnd
l2norm(:)  = norm2e( mat(:,:) , dim ) ! mat is real    array of kind stnd
l2norm(:)  = norm2e( mat(:,:) , dim ) ! mat is complex array of kind stnd
```

lassq2e()

purpose:

lassq2e() computes a scaled sum of squares based on an updated version of the LAPACK3E' algorithm [Anderson:2002] [Anderson:2018] and uses compensated summation in order to minimize rounding errors [Hanson_Hopkins:2018].

Synopsis:

```
call lassq2e( vec(:)   , scal, ssq ) ! vec is real    array of kind stnd
call lassq2e( vec(:)   , scal, ssq ) ! vec is complex array of kind stnd
call lassq2e( mat(:,:) , scal, ssq ) ! mat is real    array of kind stnd
call lassq2e( mat(:,:) , scal, ssq ) ! mat is complex array of kind stnd
```

scatter_add()

Synopsis:

```
call scatter_add( dest(:) , source(:n) , dest_index(:n) ) ! dest and sources␣
↪are integer arrays of kind i4b
call scatter_add( dest(:) , source(:n) , dest_index(:n) ) ! dest and sources␣
↪are real    arrays of kind stnd
call scatter_add( dest(:) , source(:n) , dest_index(:n) ) ! dest and sources␣
↪are complex arrays of kind stnd
```

scatter_max()

Synopsis:

```
call scatter_max( dest(:) , source(:n) , dest_index(:n) ) ! dest and sources␣
↪are integer arrays of kind i4b
call scatter_max( dest(:) , source(:n) , dest_index(:n) ) ! dest and sources␣
↪are real    arrays of kind stnd
```

diagadd()

Synopsis:

```
call diagadd( mat(:,:)   , diag            ) ! mat is a real    array of kind␣
↪stnd
call diagadd( mat(:,:)   , diag            ) ! mat is a complex array of kind␣
↪stnd
call diagadd( mat(:n,:m) , diag(:min(n,m)) ) ! diag and mat are real    ␣
↪arrays of kind stnd
call diagadd( mat(:n,:m) , diag(:min(n,m)) ) ! diag and mat are complex␣
↪arrays of kind stnd
```

diagmult()

Synopsis:

```
call diagmult( mat(:,:)   , diag            ) ! mat is a real    array of␣
↪kind stnd
call diagmult( mat(:,:)   , diag            ) ! mat is a complex array of␣
↪kind stnd
call diagmult( mat(:n,:m) , diag(:min(n,m)) ) ! diag and mat are real    ␣
↪arrays of kind stnd
call diagmult( mat(:n,:m) , diag(:min(n,m)) ) ! diag and mat are complex␣
↪arrays of kind stnd
```

get_diag()

Synopsis:

```
diag(:min(n,m)) = get_diag( mat(:n,:m) ) ! mat is a real    array of kind stnd
diag(:min(n,m)) = get_diag( mat(:n,:m) ) ! mat is a complex array of kind stnd
```

put_diag()

Synopsis:

```
call put_diag( diag             , mat(:,:)  ) ! mat is a real    array of␣
↪kind stnd
call put_diag( diag             , mat(:,:)  ) ! mat is a complex array of␣
↪kind stnd
call put_diag( diag(:min(n,m)) , mat(:n,:m) ) ! diagv and mat are real    ␣
↪arrays of kind stnd
call put_diag( diag(:min(n,m)) , mat(:n,:m) ) ! diagv and mat are complex␣
↪arrays of kind stnd
```

unit_matrix()

Synopsis:

```
call unit_matrix( mat(:,:) ) ! mat is a real    array of kind stnd
call unit_matrix( mat(:,:) ) ! mat is a complex array of kind stnd
```

Examples:

ex2_trid_inviter.F90

ex2_trid_deflate.F90

lascl()

Synopsis:

```
call lascl( x         , cfrom , cto )
call lascl( x(:)      , cfrom , cto )
call lascl( x(:,:)    , cfrom , cto )
call lascl( x(:,:)    , cfrom , cto , type         )
call lascl( x         , cfrom , cto , mask         )
call lascl( x(:n)     , cfrom , cto , mask(:n)     )
call lascl( x(:n,:m)  , cfrom , cto , mask(:n,:m)  )
```

pythag()

purpose:

Computes $\sqrt{a^2 + b^2}$ without destructive underflow or overflow.

Synopsis:

```
x = pythag( a , b )
```

pythage()

purpose:

Computes $\sqrt{a^2 + b^2}$ without destructive underflow or overflow using the Blue's scaling method [Blue:1978] [Anderson:2018].

Synopsis:

```
x = pythage( a , b )
```

5.14 MODULE Utilities_With_Pnter

Module *Utilities_With_Pnter* exports subroutines and functions for manipulating Fortran 90 pointers.

These routines are adapted from public domain routines in Numerical Recipes.

In order to use one of these routines, you must include an appropriate `use Utilities_With_Pnter` or `use Statpack` statement in your Fortran program, like:

```
use Utilities_With_Pnter, only: realloc
```

or :

```
use Statpack, only: realloc
```

Here is the list of the public routines exported by module *Utilities_With_Pnter*:

reallocate()

purpose:

reallocate() reallocates a pointer *P* to an integer, real or complex, one- or two-dimensional array with a new size *N*, while preserving its contents. The pointer *P* is deallocated on return of **reallocate()**.

Synopsis:

```
p(:n)    = reallocate( p(:)   , n     )  ! p is an allocated pointer to an␣
↪integer vector of kind i4b
```

```
p(:n)     = reallocate( p(:)    , n      )  ! p is an allocated pointer to a␣
↪real vector of kind stnd
p(:n)     = reallocate( p(:)    , n      )  ! p is an allocated pointer to a␣
↪complex vector of kind stnd
p(:n)     = reallocate( p(:)    , n      )  ! p is an allocated pointer to a␣
↪character vector
p(:n,:m) = reallocate( p(:,:) , n , m )  ! p is an allocated pointer to an␣
↪integer matrix of kind i4b
p(:n,:m) = reallocate( p(:,:) , n , m )  ! p is an allocated pointer to a␣
↪real matrix of kind stnd
p(:n,:m) = reallocate( p(:,:) , n , m )  ! p is an allocated pointer to a␣
↪complex matrix of kind stnd
```

realloc()

purpose:

realloc() reallocates a pointer *P* to an integer, real or complex, one- or two-dimensional array with a new size *N*, while preserving its contents.

Synopsis:

```
call realloc( p(:)    , n      , ialloc )  ! p is an allocated pointer to an␣
↪integer vector of kind i4b
call realloc( p(:)    , n      , ialloc )  ! p is an allocated pointer to a␣
↪real vector of kind stnd
call realloc( p(:)    , n      , ialloc )  ! p is an allocated pointer to a␣
↪complex vector of kind stnd
call realloc( p(:)    , n      , ialloc )  ! p is an allocated pointer to a␣
↪character vector
call realloc( p(:,:) , n , m , ialloc )  ! p is an allocated pointer to an␣
↪integer matrix of kind i4b
call realloc( p(:,:) , n , m , ialloc )  ! p is an allocated pointer to a␣
↪real matrix of kind stnd
call realloc( p(:,:) , n , m , ialloc )  ! p is an allocated pointer to a␣
↪complex matrix of kind stnd
```

5.15 MODULE Random

Module *Random* exports subroutines and functions for random number or array/matrix generation and related procedures.

Randomized algorithms for linear algebra computations (using random Gaussian matrix projections) are also included [Martinsson:2019] [Erichson_etal:2019]. More specifically, a large variety of very fast randomized routines for performing full or partial QR factorization with Column Pivoting (QRCP) or orthogonal factorizations of a matrix [Duersch_Gu:2017] [Martinsson_etal:2017] [Xiao_etal:2017] [Duersch_Gu:2020] are available, as well as (randomized and deterministic) routines for computing rank revealing partial QB decompositions [Martinsson:2019] [Martinsson_Voronin:2016] [Yu_etal:2018], the column Interpolative Decomposition (ID), the two-sided Interpolative Decomposition (tsID) and the CUR decomposition of a matrix [Stewart:1999] [Voronin_Martinsson:2017] [Martinsson:2019]. The ID, tsID and CUR (full or partial) decompositions are popular in data analysis as they can address some of the flaws of low-rank approximations based on the SVD, which are included and described in module *SVD_Procedures*. As an illustration, the CUR decomposition computes an approximation of an input matrix `MAT` of the form:

$$MAT \simeq C * U * R$$

where C and R contain, respectively, a reduced set of actual columns and rows of MAT, and, usually,

$$U = C^{+} * MAT * R^{+}$$

where C^{+} and R^{+} are the generalized (e.g., Moore-Penrose) inverses [Golub_VanLoan:1996] of C and R, respectively [Martinsson:2019]. As C and R are composed of actual columns and rows of MAT, they preserve the sparsity in the data and they can be more easily interpreted than the leading left and right singular vectors in the SVD of MAT, which gives the optimal low-rank approximation of MAT, as these singular vectors are linear combinations of all the columns and rows of the input matrix MAT.

Some parts of this module related to random number and array generation are adapted from [Hennecke:1995]. Note also that the successful compilation of the *Random* module (e.g., the file Module_Random.F90 in the $STATPACKDIR/sources directory) on your system may require the use of some cpp macros. See the section *Preprocessor cpp macros* for more details.

Random number generation subroutines in this module may be used to replace the Fortran 90 intrinsic routines **random_number()** and **random_seed()** by several implementations of the Marsaglia's KISS (e.g., Keep It Simple Stupid), L'Ecuyer's LFSR113, Mersenne Twister's MT19937 AND MEMT19937-II uniform random generators.

The Marsaglia's KISS (Keep It Simple Stupid) random number generator combines:

1) The congruential generator $x(n) = 69069 * x(n-1) + 1327217885$ with a period of 2^{32};
2) A 3-shift shift-register generator with a period of 2^{32} - 1;
3) Two 16-bit multiply-with-carry generators with a period of $597273182964842497 > 2^{59}$.

The overall period of this KISS random number generator exceeds 2^{123}. More details on this Marsaglia's KISS random number generator are available in [Marsaglia:1999] and [Marsaglia:2005]. This generator is also the one supplied by the intrinsic subroutine **random_number()** as implemented in previous versions of the GNU gfortran compiler.

The module also includes a "fast" version of the KISS random number generator, which uses only add, shift, exclusive-or and 'and' operations to produces exactly the same 32-bit integer output, which C views as unsigned and Fortran views as signed integers. This KISS version avoids multiplication and is probably faster. More details are available in [Marsaglia:2007].

The LFSR113 random number generator is described in [LEcuyer:1999]. This random number generator has a period length of about 2^{113}

The MT19937 Mersenne Twister random number generator is described in [Matsumoto_Nishimura:1998]. This random number generator has a period length of about 2^{19937} - 1, and 623-dimensional equidistribution property is assured. This random number generator is also the one supplied by the intrinsic subroutine **random_number()** as implemented by the NAG nagfor compiler.

The MEMT19937-II Mersenne Twister random number generator is described in [Harase:2014]. This random number generator has also a period length of about 2^{19937} - 1, and a new set of parameters is introduced in the tempering phase of MT19937, which gives a maximally equidistributed Mersenne Twister random number generator.

For all the random number generators described above, extended precision versions are also available to generate full precision random real numbers of kind **stnd** (up to 63-bit precision) using the method described in [Doornik:2007].

The choice between these 10 different uniform random generators can be done with a call to the subroutine *random_seed_()* by specifying the optional *ALG* argument (see below).

The FORTRAN versions of these random number generators as implemented here require that 32-bits integer type is available on your computer and that 32-bits integers are represented in base 2 with two's complement notation.

However, the LFSR113, MT19937 and MEMT19937-II Mersenne Twister random number generators will also work if only 64-bits integer type is available on your system, but in that case you must specify the cpp macro _RANDOM_NOINT32 at compilation of the STATPACK library. See the section *Preprocessor cpp macros* for more details. The other random number generators will not work properly with 64-bits integer type so they cannot be used on such system.

The KISS random number generators also assumed that integer overflows do not cause crashes. These assumptions are checked before using these random number generators. On the other hand, the LFSR113, MT19937 and MEMT19937-II random number generators do not use integer arithmetic and are free of such assumptions.

These 10 different uniform random generators are implemented by the routines *random_number_()* and *random_seed_()*, described below, and which also follow the standard Fortran 90 interfaces defined by the intrinsic procedures **random_number()** and **random_seed()** [Fortran]. The only exception is the addition of the optional argument *ALG* in the *random_seed_()* subroutine, which allows the user to select the random generator he wants to use subsequently in his Fortran program.

The *random_seed_()* subroutine can be used to seed the different STATPACK random generators as defined in the Fortran standard [Fortran]. Note, however, that both the different versions of the MT19937 and MEMT19937-II random number generators have a very large state (`630` 32-bit integers), and therefore it is strongly recommended that the *random_seed_()* routine only be used with a *PUT* argument that is the value returned by a previous call with a *GET* argument; i.e., only to repeat a previous sequence for these generators. This is because if a user-specified seed has low entropy (likely since there are `630` values to be supplied), it is highly likely to set these generators to an apparently-low-entropy part of their sequence.

Moreover, as the seed is used as a random bit-stream, it is expected to have approximately half of its bits nonzero. Thus, providing many small integer values will likely result in a low-entropy part of the MT19937 and MEMT19937-II sequences being reached (this is also true for the other STATPACK random generators).

If you do want to provide your own seed (and thus entropy), for the MT19937 and MEMT19937-II random number generators, it is better to use the *init_mt19937()* and *init_memt19937()* subroutines (described below), which allow you to use any size for their integer seed vector argument, but limit the risk of reaching a low-entropy part of the MT19937 and MEMT19937-II sequences.

In order to use routines *random_number_()* and *random_seed_()* provided by module *Random* instead of the intrinsic Fortran 90 procedures **random_number()** and **random_seed()** in your Fortran program, include an appropriate `use Random` (or `use Statpack`) statement, like:

```
use Random, only: random_number=>random_number_, random_seed=>random_seed_
```

In addition to these different uniform random real generators, this module also provides:

- subroutines and functions for random (signed and unsigned) integer generation;
- Gaussian random generators [Thomas_etal:2007];
- shuffling and sampling routines [Noreen:1989];
- subroutines for generating pseudo-random orthogonal matrices as Householder transformation matrices or following the Haar distribution over the group of orthogonal matrices [Stewart:1980];
- subroutines for generating pseudo-random symmetric matrices with a prescribed spectrum;
- subroutines for generating pseudo-random matrices with a prescribed singular value distribution;
- subroutines for computing a randomized (partial or full) QR factorization with Column Pivoting (QRCP) or Complete Orthogonal Decomposition (COD) of a matrix [Duersch_Gu:2020];
- subroutines for computing a rank-revealing QB decomposition of a matrix [Martinsson_Voronin:2016] [Yu_etal:2018];
- subroutines for computing column Interpolative Decomposition (ID), two-sided Interpolative Decomposition (tsID) and CUR decomposition of a matrix [Martinsson:2019].

The Gaussian random generators provided in this module use the classical Box-Muller method or the Cumulative Density Function (CDF) inversion method to generate Gaussian random real numbers of kind **stnd** or **extd** [Thomas_etal:2007].

Random generators for other probability distribution functions are not provided in this version of STATPACK, but can be easily constructed with the help of the inverse distribution functions included in the *Prob_procedures* module.

In order to use one of the routines provided by the *Random* module, you must include an appropriate `use Random` or `use Statpack` statement in your Fortran program, like:

```
use Random, only: rand_number
```

or :

```
use Statpack, only: rand_number
```

Here is the list of the public routines exported by module *Random*:

random_seed_()

Purpose:

This subroutine provides an user interface for seeding the random number routines in module *Random*.

Syntax is like **random_seed()** intrinsic subroutine and a call to **random_seed_()** without arguments initiates a non-repeatable reset of the seeds used by the random number subroutines and functions in module *Random*.

As for **random_seed()** intrinsic subroutine, no more than one argument may be specified in a call to **random_seed_()**.

Note that the size of the seed array varies according to the selected random generator (e.g., with the value of the optional *ALG* argument).

Synopsis:

```
call random_seed_( alg=alg , size=size , put=put , get=get )
```

Examples:

ex1_rsvd_cmp.F90

ex1_rqr_svd_cmp.F90

ex1_rqlp_svd_cmp.F90

ex1_rqlp_svd_cmp2.F90

ex1_random_svd.F90

ex1_random_eig.F90

ex1_random_eig_pos.F90

ex1_random_eig_with_blas.F90

ex1_random_eig_pos_with_blas.F90

init_mt19937()

Purpose:

User interface subroutine for initializing the state of the MT19937 Random Number Generator (RNG) with a scalar or vector integer seed of kind **i4b** directly, without using the subroutine *random_seed_()* and its interface.

Synopsis:

```
call init_mt19937( seed    )
call init_mt19937( seed(:) )
```

init_memt19937()

Purpose:

User interface subroutine for initializing the state of the MEMT19937-II Random Number Generator (RNG) with a scalar or vector integer seed of kind **i4b** directly, without using the subroutine *random_seed_()* and its interface.

Synopsis:

```
call init_memt19937( seed    )
call init_memt19937( seed(:) )
```

rand_number()

Purpose:

This function returns an uniformly distributed random number between 0 and 1, exclusive of the two endpoints 0 and 1.

However, if the cpp macro _RANDOM_WITH0 is used for the compilation of the STATPACK library, this function may return the zero value.

Synopsis:

```
harvest = rand_number( ) ! harvest is a real number of kind stnd
```

random_number_()

Purpose:

This generic subroutine returns an uniformly distributed random array (or number) between 0 and 1, exclusive of the two endpoints 0 and 1.

However, if the cpp macro _RANDOM_WITH0 is used for the compilation of the STATPACK library, this subroutine may return the zero value.

Synopsis:

```
call random_number_( harvest )                ! harvest is an uniform random␣
↪real number of kind stnd between 0 and 1
call random_number_( harvest(:) )             !
call random_number_( harvest(:,:) )           !
call random_number_( harvest(:,:,:) )         !
call random_number_( harvest(:,:,:,:) )       ! harvest is an uniform random␣
↪real array of kind stnd between 0 and 1
call random_number_( harvest(:,:,:,:,:) )     !
call random_number_( harvest(:,:,:,:,:,:) )   !
call random_number_( harvest(:,:,:,:,:,:,:) ) !
```

Examples:

ex1_random_number_.F90

ex1_trid_inviter.F90

ex1_trid_inviter_bis.F90

rand_integer32()

Purpose:

This function returns a random integer in the interval (-2147483648 , 2147483647) inclusive of the two endpoints.

The returned integer is equivalent to a signed 32-bit integer.

Synopsis:

```
harvest = rand_integer32( ) ! harvest is a signed 32-bit integer of kind i4b
```

random_integer32_()

Purpose:

This generic subroutine returns an array (or number) of random integers in the interval `(-2147483648 , 2147483647)` inclusive of the two endpoints.

The returned integers are equivalent to signed 32-bit integers.

Synopsis:

```
call random_integer32_( harvest )                ! harvest is an uniform␣
↪random signed 32-bit integer of kind i4b
call random_integer32_( harvest(:) )             !
call random_integer32_( harvest(:,:) )           !
call random_integer32_( harvest(:,:,:) )         !
call random_integer32_( harvest(:,:,:,:) )       ! harvest is an uniform␣
↪random signed 32-bit integer array of kind i4b
call random_integer32_( harvest(:,:,:,:,:) )     !
call random_integer32_( harvest(:,:,:,:,:,:) )   !
call random_integer32_( harvest(:,:,:,:,:,:,:) ) !
```

rand_integer31()

Purpose:

This function returns a random integer in the interval `(0 , 2147483647)` inclusive of the two endpoints.

The returned integer is equivalent to an unsigned 31-bit integer.

Synopsis:

```
harvest = rand_integer31( ) ! harvest is an unsigned (positive) 31-bit␣
↪integer of kind i4b
```

random_integer31_()

Purpose:

This generic subroutine returns an array (or number) of random integers in the interval `(0 , 2147483647)` inclusive of the two endpoints.

The returned integers are equivalent to unsigned 31-bit integers.

Synopsis:

```
call random_integer31_( harvest )                ! harvest is an uniform␣
↪random unsigned 31-bit integer of kind i4b
call random_integer31_( harvest(:) )             !
call random_integer31_( harvest(:,:) )           !
call random_integer31_( harvest(:,:,:) )         !
call random_integer31_( harvest(:,:,:,:) )       ! harvest is an uniform␣
↪random unsigned 31-bit integer array of kind i4b
call random_integer31_( harvest(:,:,:,:,:) )     !
call random_integer31_( harvest(:,:,:,:,:,:) )   !
call random_integer31_( harvest(:,:,:,:,:,:,:) ) !
```

normal_rand_number()

Purpose:

This function returns a Gaussian distributed random real number of kind **stnd**.

This function uses a Cumulative Density Function (CDF) inversion method to generate a Gaussian random real number of kind **stnd**.

Synopsis:

```
harvest = normal_rand_number( ) ! harvest is a real number of kind stnd
```

normal_random_number_()

Purpose:

This generic subroutine returns a random number or array *HARVEST* of kind **stnd** following the standard normal (Gaussian) distribution.

This subroutines uses a Cumulative Density Function (CDF) inversion method to generate Gaussian random numbers of kind **stnd**.

Synopsis:

```
call normal_random_number_( harvest )       ! harvest is a Gaussian␣
↪distributed random real number of kind stnd
call normal_random_number_( harvest(:) )    ! harvest is a Gaussian␣
↪distributed random real vector of kind stnd
call normal_random_number_( harvest(:,:) )  ! harvest is a Gaussian␣
↪distributed random real matrix of kind stnd
```

Examples:

ex1_normal_random_number_.F90

normal_rand_number2()

Purpose:

This function returns a Gaussian distributed random real number of kind **extd**.

This function uses a Cumulative Density Function (CDF) inversion method to generate a Gaussian random real number of kind **extd** and is more accurate than function *normal_rand_number()* even if kinds **extd** and **stnd** are equivalent.

Synopsis:

```
harvest = normal_rand_number2( ) ! harvest is a real number of kind extd
```

normal_random_number2_()

Purpose:

This generic subroutine returns a random number or array *HARVEST* of kind **extd** following the standard normal (Gaussian) distribution.

This subroutines uses a Cumulative Density Function (CDF) inversion method to generate Gaussian random numbers of kind **extd** and is more accurate than subroutine normal_random_number() even if kinds **extd** and **stnd** are equivalent.

Synopsis:

```
call normal_random_number2_( harvest )       ! harvest is a Gaussian␣
↪distributed random real number of kind extd
call normal_random_number2_( harvest(:) )    ! harvest is a Gaussian␣
↪distributed random real vector of kind extd
call normal_random_number2_( harvest(:,:) )  ! harvest is a Gaussian␣
↪distributed random real matrix of kind extd
```

Examples:

ex1_normal_random_number2_.F90

normal_rand_number3()

Purpose:

This function returns a Gaussian distributed random real number of kind **stnd**.

This function uses the classical Box-Muller method to generate a Gaussian random real number of kind **stnd**.

Synopsis:

```
harvest = normal_rand_number3( ) ! harvest is a real number of kind stnd
```

normal_random_number3_()

Purpose:

This generic subroutine returns a random number or array *HARVEST* of kind **stnd** following the standard normal (Gaussian) distribution.

This subroutine uses the classical Box-Muller method to generate Gaussian random real numbers of kind **stnd**.

Synopsis:

```
call normal_random_number3_( harvest )       ! harvest is a Gaussian␣
↪distributed random real number of kind stnd
call normal_random_number3_( harvest(:) )    ! harvest is a Gaussian␣
↪distributed random real vector of kind stnd
call normal_random_number3_( harvest(:,:) )  ! harvest is a Gaussian␣
↪distributed random real matrix of kind stnd
```

Examples:

ex1_normal_random_number3_.F90

ex1_random_svd.F90

ex1_random_eig.F90

ex1_random_eig_pos.F90

ex1_random_eig_with_blas.F90

ex1_random_eig_pos_with_blas.F90

random_qr_cmp()

Purpose:

This subroutine generates the first `k` columns of a pseudo-random QR factorization (in factored form) of a hypothetical real `n`-by-`n` matrix `MAT`, whose elements follow independently the standard normal distribution:

$$MAT = Q * R$$

where `Q` is a pseudo-random orthogonal matrix following the Haar distribution from the group of orthogonal matrices and `R` is upper triangular.

The upper-diagonal elements of `R` follow the standard normal distribution and the squares of the diagonal elements of `R`, `R(i,i)`^{2}, follow a chi-squared distribution with `n-i+1` degrees of freedom.

This subroutine uses a fast method based on Householder transformations for generating the first `k` columns of a `n`-by-`n` pseudo-random orthogonal matrices `Q` distributed according to the Haar measure over the orthogonal group of order `n`, in a factored form [Stewart:1980].

Synopsis:

```
call random_qr_cmp( mat(:n,:k) , diagr(:k) , beta(:k) , fillr=fillr ,␣
↪initseed=initseed )
```

ortho_gen_random_qr()

Purpose:

This subroutine generates the first k columns of a n-by-n real pseudo-random orthogonal matrix following the Haar distribution, which is defined as the product of n elementary Householder reflectors of order n and of a n-by-n diagonal matrix with diagonal elements equal to sign(one, DIAGR) [Stewart:1980]:

$$Q = H(1) * H(2) * ... * H(n) * diag(sign(DIAGR))$$

as returned by subroutine *random_qr_cmp()*.

Synopsis:

```
call ortho_gen_random_qr( mat(:n,:k) , diagr(:k) , beta(:k) )
```

gen_random_sym_mat()

Purpose:

This subroutine generates a pseudo-random n-by-n real symmetric matrix with prescribed eigenvalues.

Optionally, the corresponding eigenvectors of the generated pseudo-random n-by-n real symmetric matrix can be computed if required.

Synopsis:

```
call gen_random_sym_mat( eigval(:k) , mat(:n,:n) , eigvec=eigvec(:n,:k) ,␣
↪initseed=initseed , hous=hous )
```

Examples:

ex1_gen_random_sym_mat.F90

ex1_trid_inviter.F90

ex1_trid_inviter_bis.F90

ex1_random_eig.F90

ex1_random_eig_pos.F90

ex1_random_eig_with_blas.F90

ex1_random_eig_pos_with_blas.F90

gen_random_mat()

Purpose:

This subroutine generates a pseudo-random m-by-n real matrix with prescribed singular values.

Optionally, the corresponding singular vectors of the generated pseudo-random m-by-n real matrix can be computed if required.

Synopsis:

```
call gen_random_mat( eigval(:k) , mat(:m,:n) , leftvec=leftvec(:m,:k) ,␣
↪rightvec=rightvec(:n,:k) , initseed=initseed , hous=hous )
```

Examples:

ex1_gen_random_mat.F90

ex1_rsvd_cmp.F90

ex1_rqr_svd_cmp.F90

ex1_rqlp_svd_cmp.F90

ex1_rqlp_svd_cmp2.F90

ex1_random_svd.F90

ex1_random_svd_with_blas.F90

ex1_random_svd_fixed_precision_with_blas.F90

gen_random_ortho_mat()

Purpose:

This function generates a pseudo-random m-by-n real matrix with with orthonormal columns. This pseudo-random m-by-n real matrix with orthonormal columns is generated as the first n columns of a m-by-m Householder matrix H.

Synopsis:

```
mat(:m,:n) = gen_random_ortho_mat( m , n , initseed=initseed )
```

Examples:

ex1_gen_random_ortho_mat.F90

partial_rqr_cmp()

Purpose:

partial_rqr_cmp() computes a (partial or full) QRCP or COD factorization of a real m-by-n matrix MAT using randomized techniques based on random Gaussian matrix projections [Duersch_Gu:2017] [Martinsson_etal:2017] [Xiao_etal:2017] [Duersch_Gu:2020]. MAT may be rank-deficient.

partial_rqr_cmp() performs the same task as *qr_cmp2()* and *partial_qr_cmp()* in module *QR_Procedures*, but are much faster, and also slightly less accurate, because of the use of randomization for selecting the pivot columns in the QRCP. The use of randomization for pivot selection in the QRCP algorithm allows to perform most of the parts of the algorithm mainly with matrix-matrix operations (e.g., "BLAS3") as in the simple QR factorization without column pivoting (see the *qr_cmp()* subroutine in module *QR_Procedures* for more details).

The routine first computes a QRCP of MAT:

$$MAT * P = Q * R$$

here P is n-by-n permutation matrix, which is computed by randomized techniques [Duersch_Gu:2017] [Xiao_etal:2017], R is an upper triangular or trapezoidal (if n>m) matrix and Q is a m-by-m orthogonal matrix.

At the user option, the QR factorization can be only partial, e.g., the subroutine ends when the numbers of selected columns of MAT is equal to a predefined value equals to kpartial = size(DIAGR) = size(BETA).

This leads implicitly to the following partition of Q:

$$Q = \begin{bmatrix} Q1 & Q2 \end{bmatrix}$$

where Q1 is a m-by-kpartial orthonormal matrix and Q2 is a m-by-(m−kpartial) orthonormal matrix orthogonal to Q1, and to the following corresponding partition of R:

$$R = \begin{bmatrix} R11 & R12 \\ 0 & R22 \end{bmatrix}$$

where R11 is a kpartial-by-kpartial triangular matrix, R21 is zero by construction, R12 is a full kpartial-by-(n−kpartial) matrix and R22 is a full (m−kpartial)-by-(n−kpartial) matrix.

From these partitions of Q and R, we can obtain a good approximation of MAT of rank kpartial, since:

$$MAT * P \simeq Q1 * \begin{bmatrix} R11 & R12 \end{bmatrix}$$

and, finally:

$$MAT \simeq Q1 * \begin{bmatrix} R11 & R12 \end{bmatrix} * P^T$$

which is equivalent to assume that R22 is negligible.

If *TAU* is present, R12 is then annihilated by orthogonal transformations from the right, arriving at the partial COD:

$$MAT * P \simeq Q1 * \begin{bmatrix} T11 & 0 \end{bmatrix} * Z$$

where P is a n-by-n permutation matrix, Q1 is a m-by-kpartial orthonormal matrix, Z is a n-by-n orthogonal matrix and T11 is a kpartial-by-kpartial upper triangular matrix.

As in subroutine *partial_qr_cmp()*, if the optional argument *TOL* is present, calculations to determine the 1-norm condition number of R11 are performed and this condition number is used to determine the effective pseudo-rank of R11, krank. If this effective pseudo-rank is less than kpartial, which implies that the rank of MAT is also less than kpartial, the subroutine outputs a partial QR factorization corresponding to this effective pseudo-rank krank, instead of rank kpartial.

In all cases, the subroutine outputs krank (or kpartial if *TOL* is absent) in the argument *KRANK* and $||MAT(krank+1:m, krank+1:n)||_F$ gives the error of the associated matrix approximation in the Frobenius norm, on exit.

Finally, note that **partial_rqr_cmp()** is an effective and efficient way for computing a low-rank approximation of MAT, but is less effective than *partial_qr_cmp()* to find the rank of MAT because of the use of randomization. As an illustration, the diagonal elements of R11 are not necessarily of decreasing absolute magnitude when computed by **partial_rqr_cmp()**, while this property is enforced with *partial_qr_cmp()*. See [Martinsson_etal:2017] [Xiao_etal:2017] for details. On the other hand, *partial_qr_cmp()* can be used safely for both tasks, but is much slower than **partial_rqr_cmp()**.

Synopsis:

```
call partial_rqr_cmp( mat(:m,:n) , diagr(:kpartial) , beta(:kpartial) ,␣
↪ip(:n) , krank , tol=tol , tau=tau(:kpartial) , rng_alg=rng_alg , blk_
↪size=blk_size , nover=nover )
```

Examples:

ex1_partial_rqr_cmp.F90

ex2_partial_rqr_cmp.F90

ex3_partial_rqr_cmp.F90

partial_rqr_cmp2()

Purpose:

partial_rqr_cmp2() computes a (partial or full) QRCP or COD factorization of a real m-by-n matrix MAT using randomized techniques based on random Gaussian matrix projections [Duersch_Gu:2017] [Martinsson_etal:2017] [Xiao_etal:2017] [Duersch_Gu:2020]. MAT may be rank-deficient.

In other words, **partial_rqr_cmp2()** performs the same task as *partial_rqr_cmp()* above. The main difference between the two subroutines is in the randomized technique for the pivot selection and the computation of the permutation matrix P. *partial_rqr_cmp()* uses an efficient updating formulae to recompute the compression matrix at each iteration (see [Duersch_Gu:2017] and [Xiao_etal:2017] for details) while **partial_rqr_cmp2()** regenerates the Gaussian and compression matrices at each iteration of the randomized (partial or full) blocked QRCP algorithm. This implies that *partial_rqr_cmp()* is usually faster than **partial_rqr_cmp2()** for very large matrices, but may be slightly less accurate in some cases.

The routine first computes a QRCP of MAT:

$$MAT * P = Q * R$$

here P is n-by-n permutation matrix, which is computed by randomized techniques [Duersch_Gu:2017] [Xiao_etal:2017], R is an upper triangular or trapezoidal (if n>m) matrix and Q is a m-by-m orthogonal matrix.

At the user option, the QRCP factorization can be only partial, e.g., the subroutine ends when the numbers of selected columns of MAT is equal to a predefined value equals to kpartial = size(DIAGR) = size(BETA).

This leads implicitly to the following partition of Q:

$$Q = \begin{bmatrix} Q1 & Q2 \end{bmatrix}$$

where Q1 is a m-by-kpartial orthonormal matrix and Q2 is a m-by-(m−kpartial) orthonormal matrix orthogonal to Q1, and to the following corresponding partition of R:

$$R = \begin{bmatrix} R11 & R12 \\ 0 & R22 \end{bmatrix}$$

where R11 is a kpartial-by-kpartial triangular matrix, R21 is zero by construction, R12 is a full kpartial-by-(n−kpartial) matrix and R22 is a full (m−kpartial)-by-(n−kpartial) matrix.

From these partitions of Q and R, we can obtain a good approximation of MAT of rank kpartial, since:

$$MAT * P \simeq Q1 * \begin{bmatrix} R11 & R12 \end{bmatrix}$$

and, finally:

$$MAT \simeq Q1 * \begin{bmatrix} R11 & R12 \end{bmatrix} * P^T$$

which is equivalent to assume that R22 is negligible.

If *TAU* is present, R12 is then annihilated by orthogonal transformations from the right, arriving at the partial COD factorization:

$$MAT * P \simeq Q1 * \begin{bmatrix} T11 & 0 \end{bmatrix} * Z$$

where P is a n-by-n permutation matrix, Q1 is a m-by-kpartial orthonormal matrix, Z is a n-by-n orthogonal matrix and T11 is a kpartial-by-kpartial upper triangular matrix.

As in subroutine *partial_qr_cmp()*, if the optional argument *TOL* is present, calculations to determine the 1-norm condition number of R11 are performed and this condition number is used to determine the effective pseudo-rank of R11, krank. If this effective pseudo-rank is less than kpartial, which implies that the rank of MAT is also less than kpartial, the subroutine outputs a partial QR factorization corresponding to this effective pseudo-rank krank, instead of rank kpartial.

In all cases, the subroutine outputs krank (or kpartial if *TOL* is absent) in the argument *KRANK* and $||MAT(krank+1:m, krank+1:n)||_F$ gives the error of the associated matrix approximation in the Frobenius norm, on exit.

Finally, note that **partial_rqr_cmp2()** is an effective and efficient way for computing a low-rank approximation of MAT, but is less effective than *partial_qr_cmp()* to find the rank of MAT because of the use of randomization. As an illustration, the diagonal elements of R11 are not necessarily of decreasing absolute magnitude when computed by **partial_rqr_cmp2()**, while this property is enforced with *partial_qr_cmp()*. See [Martinsson_etal:2017] [Xiao_etal:2017] for details. On the other hand, *partial_qr_cmp()* can be used safely for both tasks, but is much slower than **partial_rqr_cmp2()**.

Synopsis:

```
call partial_rqr_cmp2( mat(:m,:n) , diagr(:kpartial) , beta(:kpartial) ,␣
↪ip(:n) , krank , tol=tol , tau=tau(:kpartial) , rng_alg=rng_alg , blk_
↪size=blk_size , nover=nover )
```

Examples:

ex1_partial_rqr_cmp2.F90

ex2_partial_rqr_cmp2.F90

ex3_partial_rqr_cmp2.F90

partial_rtqr_cmp()

Purpose:

partial_rtqr_cmp() computes an approximate partial and truncated QRCP factorization (or COD factorization) of a real m-by-n matrix MAT using randomization techniques:

$$MAT * P \simeq Q * R$$

here P is n-by-n permutation matrix, which is computed by randomized techniques [Mary_etal:2015], R is an upper triangular or trapezoidal kpartial-by-n matrix and Q is a m-by-kpartial matrix with orthonormal columns.

The randomized QRCP factorization is only partial and truncated, e.g., the subroutine ends when the numbers of columns of Q is equal to a predefined value equals to kpartial = size(DIAGR) = size(BETA). This leads implicitly to the following partition of R:

$$R = \begin{bmatrix} R11 & R12 \end{bmatrix}$$

where R11 is a kpartial-by-kpartial triangular matrix and R12 is a full kpartial-by-(n-kpartial) matrix.

From this approximate partial and truncated QRCP factorization of MAT, **partial_rtqr_cmp()** can also estimate a partial and truncated orthogonal factorization of MAT. Thus, **partial_rtqr_cmp()** performs exactly the same tasks as *partial_rqr_cmp()* and *partial_rqr_cmp2()* subroutines and the arguments of **partial_rtqr_cmp()** are nearly the same as those in these two subroutines. However, **partial_rtqr_cmp()** is significantly faster when kpartial is relatively small and MAT is a very large matrix.

This is due to the fact that **partial_rtqr_cmp()** computes Q (in factored form), R11 and R12, but not R22 (where R22 is the bottom left (m-kpartial)-by-(n-kpartial) submatrix in the QR factorization of MAT) as *partial_rqr_cmp()* and *partial_rqr_cmp2()* subroutines. Furthermore, only an estimate of R12 is computed by **partial_rtqr_cmp()**, using a randomized algorithm described in [Mary_etal:2015], while the computation of R12 is exact in *partial_rqr_cmp()* and *partial_rqr_cmp2()* subroutines. Note also that **partial_rtqr_cmp()** does not recompute or update the compression matrix at each iteration as *partial_rqr_cmp()* and *partial_rqr_cmp2()*. See [Mary_etal:2015] [Duersch_Gu:2017] [Martinsson_etal:2017] [Xiao_etal:2017] [Duersch_Gu:2020] for more information.

All these features explain why **partial_rtqr_cmp()** is usually much faster than *partial_rqr_cmp()* and *partial_rqr_cmp2()* subroutines, but is also less accurate, especially for matrices with a slow decay of their singular values. See [Mary_etal:2015] for details.

Synopsis:

```
call partial_rtqr_cmp( mat(:m,:n) , diagr(:kpartial) , beta(:kpartial)␣
↪, ip(:n) , krank , tol=tol , tau=tau(:kpartial) , rng_alg=rng_alg ,␣
↪niter=niter , nover=nover )
```

Examples:

ex1_partial_rtqr_cmp.F90

partial_rqr_cmp_fixed_precision()

Purpose:

partial_rqr_cmp_fixed_precision() computes a partial QRCP or COD factorization of a real m-by-n matrix MAT using randomized techniques:

$$MAT * P \simeq Q * R$$

here P is a n-by-n permutation matrix computed using randomization techniques, R is a kank-by-n upper triangular or trapezoidal matrix and Q is a m-by-krank with orthonormal columns. This leads to the following matrix approximation of MAT of rank krank:

$$MAT \simeq Q * R * P^T$$

krank is the target rank of the matrix approximation, which is sought, and this partial factorization must have an approximation error which fulfills:

$$||MAT - Q * R * P^T||_F <= ||MAT||_F * relerr$$

where $||||_F$ is the Frobenius norm and relerr is a prescribed accuracy tolerance for the relative error of the computed matrix approximation, specified in the input argument *RELERR*.

Thus, **partial_rqr_cmp_fixed_precision()** performs the same task as *partial_rqr_cmp()* and *partial_rqr_cmp2()*, but allows to stop the factorization at any stage in order to obtain a partial QRCP (or COD) factorization of MAT, which fullfills the above inequality.

In other words, krank, the rank of the matrix approximation, is not known in advance and is computed by the subroutine, while krank is fixed a priori and is an input argument in *partial_rqr_cmp()* and *partial_rqr_cmp2()* subroutines. Otherwise, all other arguments of **partial_rqr_cmp_fixed_precision()** have the same meaning as in *partial_rqr_cmp()* and *partial_rqr_cmp2()*.

In all cases, on exit of **partial_rqr_cmp_fixed_precision()**, $||MAT(krank+1:m, krank+1:n)||_F$ gives the error of the associated matrix approximation in the Frobenius norm and the associated relative error in the Frobenius norm is output in argument *RELERR*.

Note finally that **partial_rqr_cmp_fixed_precision()** performs exactly the same task as *partial_qr_cmp_fixed_precision()* subroutine in module *QR_Procedures*, but is much faster on large matrices because of the use of a randomized and blocked "BLAS3" algorithm instead of a standard "BLAS2" algorithm in *partial_qr_cmp_fixed_precision()*. Another difference is that, in **partial_rqr_cmp_fixed_precision()**, the rank of the matrix approximation is increased progressively of *BLK_SIZE* by *BLK_SIZE* until the prescribed tolerance for the relative error is satisfied while, in *partial_qr_cmp_fixed_precision()*, the rank of the matrix approximation is increased one by one until the prescribed tolerance for the relative error is satisfied. In other words, the rank of the matrix approximation found by **partial_rqr_cmp_fixed_precision()** is always larger than the one found by *partial_qr_cmp_fixed_precision()* and is a multiple of *BLK_SIZE*.

Synopsis:

```
call partial_rqr_cmp_fixed_precision( mat(:m,:n) , relerr , diagr(:min(m,n)) ,
↪ beta(:min(m,n)) , ip(:n) , krank , tau=tau(:min(m,n)) , rng_alg=rng_alg ,␣
↪blk_size=blk_size , nover=nover )
```

Examples:

ex1_partial_rqr_cmp_fixed_precision.F90

rqb_cmp()

Purpose:

rqb_cmp() computes a (partial or full) QB factorization of a real m-by-n matrix MAT using randomized power or subspace iteration techniques [Halko_etal:2011] [Gu:2015] [Martinsson:2019]:

$$MAT \simeq Q * B$$

Here, Q is a m-by-nqb orthonormal matrix, B is a nqb-by-n and the product $Q * B$ is a good approximation of MAT according to the spectral or Frobenius norm. nqb is the target rank of the partial QB decomposition, which is sought, and is equal to the number of columns of the output real matrix argument Q, i.e., nqb = size(Q, 2).

At the user option, an approximate QR (eventually with column pivoting) or COD factorization of MAT can be derived from this initial QB factorization with the help of optional arguments in **rqb_cmp()** subroutine.

Synopsis:

```
call rqb_cmp( mat(:m,:n) , q(:m,:nqb) , b(:nqb,:n) , niter=niter , rng_
↪alg=rng_alg , ortho=ortho , comp_qr=com_qr , ip=ip(:n) , tol=tol ,␣
↪tau=tau(:nqb) )
```

Examples:

ex1_rqb_cmp.F90

ex1_rqb_solve.F90

rqb_cmp_fixed_precision()

Purpose:

rqb_cmp_fixed_precision() computes a partial QB factorization of a real m-by-n matrix MAT using randomized power or subspace iteration techniques [Halko_etal:2011] [Gu:2015] [Martinsson_Voronin:2016] [Yu_etal:2018] [Martinsson:2019]:

$$MAT \simeq Q * B$$

Here, Q is a m-by-nqb orthonormal matrix, B is a nqb-by-n and the product $Q * B$ is a good approximation of MAT according to the spectral or Frobenius norm. nqb is the target rank of the partial QB decomposition, which is sought, and this partial factorization must have an approximation error which fulfills:

$$||MAT - Q * B||_F <= ||MAT||_F.relerr$$

where $||||_F$ is the Frobenius norm and relerr is a prescribed accuracy tolerance for the relative error of the computed partial QB approximation in the Frobenius norm, specified as an argument (e.g., argument *RELERR*) in the call to **rqb_cmp_fixed_precision**().

In other words, nqb is not known in advance and is determined in the subroutine. This explains why the output real array arguments *Q* and *B*, which contain the computed partial QB factorization, must be declared in the calling program as pointers.

rqb_cmp_fixed_precision() searches incrementally the best (e.g., smallest) partial QB approximation, which fulfills the prescribed accuracy tolerance for the relative error based on an improved version of the randQB_FP algorithm described in [Yu_etal:2018]. See also [Martinsson_Voronin:2016]. More precisely, the rank of the partial QB approximation is increased progressively of *BLK_SIZE* by *BLK_SIZE* until the prescribed accuracy tolerance is satisfied and then improved and adjusted precisely by additional subspace iterations (as specified by the optional *NITER_QB* integer argument) to obtain the smallest partial QB approximation, which satisfies the prescribed tolerance.

Note that the product of the two integer arguments *BLK_SIZE* and *MAXITER_QB* (see below for their precise meaning), determines the maximum allowable rank of the matrix approximation, which is sought.

On exit, nqb = size(Q, 2), e.g., nqb is equal to the number of columns of the output real matrix pointer argument *Q*, which contains the computed orthonormal matrix Q and the relative error in the Frobenius norm of the computed partial QB approximation is output in argument *RELERR*.

At the user option, an approximate QR (eventually with column pivoting) or COD factorization of MAT can be derived from this initial QB factorization with the help of optional arguments in **rqb_cmp_fixed_precision**() subroutine.

Note, finally, that if you already know the rank of the partial QB approximation of MAT you are seeking, it is better to use *rqb_cmp()* rather than **rqb_cmp_fixed_precision**() as *rqb_cmp()* is faster and slightly more accurate.

Synopsis:

```
call rqb_cmp_fixed_precision( mat(:m,:n) , relerr , q(:,:) , b(:,:) , failure_
↪relerr=failure_relerr , niter=niter , rng_alg=rng_alg , blk_size=blk_size ,␣
↪maxiter_qb=maxiter_qb , ortho=ortho , reortho=reortho , niter_qb=niter_qb ,␣
↪comp_qr=com_qr , ip=ip(:n) , tol=tol , tau=tau(:nqb) )
```

Examples:

ex1_rqb_cmp_fixed_precision.F90

id_cmp()

Purpose:

id_cmp() computes a (partial) column Interpolative Decomposition (ID) of a m-by-n real matrix MAT. A column ID factorization of rank krank approximates MAT as:

$$MAT \simeq C * V$$

where C is an m-by-krank matrix, which consists of a subset of krank columns of MAT and V is a krank-by-n matrix, which contains a krank-by-krank identity matrix as a submatrix. The subset of the columns of MAT, which forms C, is selected to give a good approximation of MAT in the spectral or Frobenius norm.

Such column ID factorization can be computed with the help of a (deterministic or randomized) partial QRCP factorization of MAT. See the detailed description of **id_cmp()** and [Stewart:1999] [Berry_etal:2005] [Voronin_Martinsson:2015] [Voronin_Martinsson:2017] [Martinsson:2019] for more information.

Synopsis:

```
call id_cmp( mat(:m,:n) , ip(:n) , t(:krank,:n-krank) , c=c(:krank,:n) ,␣
↪v=v(:krank,:n) , rnorm=rnorm , diagr=diagr(:krank) , beta=beta(:krank)␣
↪, tol=tol , random_qr=random_qr , rng_alg=rng_alg , blk_size=blk_size ,␣
↪nover=nover )
```

Examples:

ex1_id_cmp.F90

ts_id_cmp()

Purpose:

ts_id_cmp() computes a (partial) two-sided Interpolative Decomposition (tsID) of a m-by-n real matrix MAT. A tsID factorization of rank krank approximates a real matrix MAT as a triple matrix product:

$$MAT \simeq W * MAT_skel * V$$

where W is an m-by-krank matrix, V is a krank-by-n matrix and MAT_skel is a squared krank-by-krank matrix, the so-called "skeleton" of MAT. W, V and MAT_skel are estimated to give a good approximation of MAT in the spectral or Frobenius norm.

Such tsID factorization can be computed with the help of a (deterministic or randomized) partial QRCP factorization of MAT and of a matrix derived from its partial QR decomposition, more precisely with a column ID of MAT and a row ID of a subset of the columns of MAT. See the detailed description of **ts_id_cmp()** and [Stewart:1999] [Berry_etal:2005] [Voronin_Martinsson:2015] [Voronin_Martinsson:2017] [Martinsson:2019] for more information.

Synopsis:

```
call ts_id_cmp( mat(:m,:n) , ip_row(:m) , ip_col(:n) , w(:m,:krank) ,␣
↪v=v(:krank,:n) , skelmat(:krank,:krank) , rnorm=rnorm , diagr=diagr(:krank)␣
↪, beta=beta(:krank) , tol=tol , random_qr=random_qr , rng_alg=rng_alg , blk_
↪size=blk_size , nover=nover )
```

Examples:

ex1_ts_id_cmp.F90

cur_cmp()

Purpose:

cur_cmp() computes a (partial) CUR Decomposition (tsID) of a m-by-n real matrix MAT. A CUR factorization of rank krank approximates a real matrix MAT as a triple matrix product:

$$MAT \simeq C * U * R$$

where C and R are m-by-krank and krank-by-n matrices, which are, respectively, subsets of the columns and rows of MAT, and U is a krank-by-krank, which is estimated to make the matrix product $C * U * R$ a good approximation of MAT according to the Frobenius norm. The CUR factorization is an important tool for handling large-scale data sets, offering several advantages over the Singular Value Decomposition (SVD): the columns and rows that comprise C and R are representative of the data and they are sparse if MAT is sparse. See [Mahoney_Drineas:2009] and [Martinsson:2019] for a discussion.

Computing an approximate CUR decomposition is generally a three-step process. The C and R matrix factors in the CUR factorization can be first estimated with the help of (randomized or deterministic) partial QRCP factorizations of MAT and MAT^T, respectively. In a final step, we then seek a krank-by-krank matrix U such that:

$$||MAT - C * U * R||_F = min$$

See the detailed description of **cur_cmp()** and [Stewart:1999] [Berry_etal:2005] [Voronin_Martinsson:2017] [Martinsson:2019] for more information on the algorithm used to estimate the matrices C, U and R of the CUR factorization.

Synopsis:

```
call cur_cmp( mat(:m,:n) , ip_row(:m) , ip_col(:n) , u(:krank,:krank) ,␣
↪c=c(:m,:krank) , r=r(:krank,:n) , rnorm_row=rnorm_row , rnorm_col=rnorm_
↪col , tol=tol , random_qr=random_qr , rng_alg=rng_alg , blk_size=blk_size ,␣
↪nover=nover )
```

Examples:

ex1_cur_cmp.F90

simple_shuffle()

Purpose:

This generic subroutine shuffles all the elements of the vector VEC.

Synopsis:

```
call simple_shuffle( vec(:) ) ! vec is a real     vector of kind stnd
call simple_shuffle( vec(:) ) ! vec is a complex  vector of kind stnd
call simple_shuffle( vec(:) ) ! vec is an integer vector of kind i4b
```

drawsample()

Purpose:

This subroutine may be used to draw a sample, without replacement of size *NSAMPLE* from a population of size SIZE(POP). On output, the integer vector POP(1:NSAMPLE) indicates which observations are included in the sample.

The integer vector POP must be dimensioned at least as large as *NSAMPLE* in the calling program.

Synopsis:

```
call drawsample( nsample , pop(:) ) ! pop is an integer vector of kind i4b
```

Examples:

ex1_drawsample.F90

ex2_drawsample.F90

drawbootsample()

Purpose:

This subroutine may be used to draw a bootstrap random sample of size SIZE(SAMPLE) from a finite population of size *NPOP*. On output, the integer vector SAMPLE indicates which observations are included in the bootstrap sample.

The sampling is done with replacement, meaning that the sample may contain duplicate observations.

Synopsis:

```
call drawbootsample( npop , sample(:) ) ! sample is an integer vector of kind␣
↪i4b
```

5.16 MODULE Giv_Procedures

Module *Giv_Procedures* exports subroutines for computing and applying Givens rotations and reflections [Golub_VanLoan:1996]. Both standard and fast Givens rotations/reflections are implemented in this module.

A Givens rotation is a rotation in the plane acting on two elements of a given vector. Givens rotations are typically used to introduce zeros in vectors, such as during the QR decomposition of a matrix [Golub_VanLoan:1996]. In this case, it is typically desired to find scalars `cs` and `sn` such that

$$\begin{pmatrix} cs & sn \\ -sn & cs \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix}$$

where $r = \sqrt{a^2 + b^2}$ and $cs^2 + sn^2 = 1$.

The standard Givens rotations/reflections routines in *Giv_Procedures* use algorithms and guidelines provided in Section 3.4 of [Anderson_Fahey:1997].

The fast Givens rotations/reflections routines in *Giv_Procedures* are implementations of the two-way branch algorithms (fast plane rotations with dynamic scaling to avoid overflow/underflow) described in [Anda_Park:1994].

Please note that routines provided in this module apply only to real data of kind **stnd**. The real kind type **stnd** is defined in module *Select_Parameters*.

In order to use one of these routines, you must include an appropriate `use Giv_Procedures` or `use Statpack` statement in your Fortran program, like:

```
use Giv_Procedures, only: define_rot_givens
```

or :

```
use Statpack, only: define_rot_givens
```

Here is the list of the public routines exported by module *Giv_Procedures*:

`define_rot_givens``()`

purpose:

define_rot_givens() generates the cosine and sine of a Givens plane rotation, so that

$$\begin{pmatrix} cs & sn \\ -sn & cs \end{pmatrix} * \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix}$$

where $r = \sqrt{a^2 + b^2}$ and $cs^2 + sn^2 = 1$.

On output, the rotation is also stored in compact form in *B* and can be recovered by the following algorithm:

- If $B = 1$, set $cs = 0$ and $sn = 1$
- If $|B| < 1$, set $sn = B$ and $cs = \sqrt{1 - sn^2}$
- If $|B| > 1$, set $cs = 1/B$ and $sn = \sqrt{1 - cs^2}$

Synopsis:

```
call define_rot_givens( a , b , cs , sn )
```

rot_givens()

purpose:

rot_givens() generates and applies a Givens plane rotation to the vector `(a b)` or to the `n`-by-`2` matrix `[VECA VECB]`, so that

$$\begin{pmatrix} cs & sn \\ -sn & cs \end{pmatrix} * \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix}$$

where $r = \sqrt{a^2 + b^2}$ and $cs^2 + sn^2 = 1$, or

$$\begin{pmatrix} cs & sn \\ -sn & cs \end{pmatrix} * \begin{bmatrix} VECA^T \\ VECB^T \end{bmatrix} = \begin{bmatrix} cs * VECA^T + sn * VECB^T \\ -sn * VECA^T + cs * VECB^T \end{bmatrix}$$

where:

- $cs^2 + sn^2 = 1$,
- $cs * VECA(1) + sn * VECB(1) = \sqrt{VECA(1)^2 + VECB(1)^2}$,
- $-sn * VECA(1) + cs * VECB(1) = 0$.

On output, the rotation is also stored in compact form in *B* (or *VECB(1)*) and can be recovered by the following algorithm:

- If $B = 1$, set $cs = 0$ and $sn = 1$
- If $|B| < 1$, set $sn = B$ and $cs = \sqrt{1 - sn^2}$
- If $|B| > 1$, set $cs = 1/B$ and $sn = \sqrt{1 - cs^2}$

Synopsis:

```
call rot_givens( a         , b                    )
call rot_givens( veca(:n) , vecb(:n)              )
call rot_givens( a         , b         , cs , sn )
call rot_givens( veca(:n) , vecb(:n) , cs , sn )
```

apply_rot_givens()

purpose:

apply_rot_givens(), eventually reconstructs a Givens plane rotation, stored in compact form in *B*, and applies this Givens plane rotation to the vector `(c d)` or to two vectors `VECC` and `VECD`.

That is, the value *B* allows the cosine and sine of the Givens plane rotation to be recovered by the following algorithm:

- If $B = 1$, set $cs = 0$ and $sn = 1$
- If $|B| < 1$, set $sn = B$ and $cs = \sqrt{1 - sn^2}$
- If $|B| > 1$, set $cs = 1/B$ and $sn = \sqrt{1 - cs^2}$

Next, the Givens plane rotation is applied to the vector `(c d)`:

$$\begin{pmatrix} cs & sn \\ -sn & cs \end{pmatrix} * \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} cs * c + sn * d \\ -sn * c + cs * d \end{pmatrix}$$

or to two vectors `VECC` and `VECD`:

$$\begin{pmatrix} cs & sn \\ -sn & cs \end{pmatrix} * \begin{bmatrix} VECC^T \\ VECD^T \end{bmatrix} = \begin{bmatrix} cs * VECC^T + sn * VECD^T \\ -sn * VECC^T + cs * VECD^T \end{bmatrix}$$

where $cs^2 + sn^2 = 1$.

Synopsis:

```
call apply_rot_givens( c        , d        , b       )
call apply_rot_givens( vecc(:n) , vecd(:n) , b       )
call apply_rot_givens( c        , d        , cs , sn )
call apply_rot_givens( vecc(:n) , vecd(:n) , cs , sn )
```

givens_vec()

purpose:

givens_vec() defines and applies a Givens plane rotation to the n-by-2 matrix [VECA VECB]. The rotation is designed to annihilate the first element of VECB (e.g., VECB(1)). That is,

$$\begin{pmatrix} cs & sn \\ -sn & cs \end{pmatrix} * \begin{bmatrix} VECA^T \\ VECB^T \end{bmatrix} = \begin{bmatrix} cs * VECA^T + sn * VECB^T \\ -sn * VECA^T + cs * VECB^T \end{bmatrix}$$

where:

- $cs^2 + sn^2 = 1$,
- $-sn * VECA(1) + cs * VECB(1) = 0$,
- $cs * VECA(1) + sn * VECB(1) = \sqrt{VECA(1)^2 + VECB(1)^2}$.

Synopsis:

```
call givens_vec( veca(:n) , vecb(:n)          )
call givens_vec( veca(:n) , vecb(:n) , cs , sn )
```

givens_mat_left()

purpose:

givens_mat_left() transforms the matrix MAT to upper trapezoidal form by applying a series of Givens plane rotations on the rows of MAT.

Synopsis:

```
call givens_mat_left( mat(:,:) )
```

Examples:

ex1_givens_mat_left.F90

givens_mat_right()

purpose:

givens_mat_right() transforms the matrix MAT to lower trapezoidal form by applying a series of Givens plane rotations on the columns of MAT.

Synopsis:

```
call givens_mat_right( mat(:,:) )
```

Examples:

ex1_givens_mat_right.F90

givens_vec_mat_left()

purpose:

givens_vec_mat_left() defines and applies a series of Givens rotations on a n-vector VEC and on the rows of a p-by-n matrix MAT. The rotations are designed to annihilate all the elements of the first column of MAT.

Synopsis:

```
call givens_vec_mat_left( vec(:n) , mat(:p,:n) )
```

givens_vec_mat_right()

purpose:

givens_vec_mat_right() defines and applies a series of Givens rotations on a `n`-vector `VEC` and on the columns of a `p`-by-`n` matrix `MAT`. The rotations are designed to annihilate all the elements of the first row of `MAT`.

Synopsis:

```
call givens_vec_mat_right( vec(:p) , mat(:p,:n) )
```

define_rot_fastgivens()

purpose:

define_rot_fastgivens() generates a fast Givens plane rotation `H` (defined by *BETA*, *ALPHA*, and *TYPE_ROT* on output) and updated scale factors (*D1* and *D2*), which zero *X2*. That is,

$$\begin{pmatrix} x1 & x2 \end{pmatrix} * H = \begin{pmatrix} r & 0 \end{pmatrix}$$

where `H` is equal to

- $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, if *TYPE_ROT* = 0.
- $\begin{pmatrix} 1 & 0 \\ B & 1 \end{pmatrix} \begin{pmatrix} 1 & A \\ 0 & 1 \end{pmatrix}$, if *TYPE_ROT* = 1
- $\begin{pmatrix} 1 & A \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ B & 1 \end{pmatrix}$, if *TYPE_ROT* = 2
- $\begin{pmatrix} 0 & -1 \\ 1 & A \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -B & 1 \end{pmatrix}$, if *TYPE_ROT* = 3
- $\begin{pmatrix} B & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & A \\ 0 & -1 \end{pmatrix}$, if *TYPE_ROT* = 4

On output, the arguments *BETA* = `B` and *ALPHA* = `A` and *TYPE_ROT* define the transformation matrix `H`:

$$H = \begin{pmatrix} h11 & h12 \\ h21 & h22 \end{pmatrix}$$

Furthermore, if on input, `y1 = x1*sqrt(d1)` and `y2 = x2*sqrt(d2)`, then on output, with the updated scale factors *D1* and *D2*:

$$\begin{pmatrix} x1 & x2 \end{pmatrix} * H * diag(\sqrt{d1} \quad \sqrt{d2}) = \begin{pmatrix} [x1 * h11 + x2 * h21] * \sqrt{d1} & 0 \end{pmatrix}$$

is equal to

$$\begin{pmatrix} y1 & y2 \end{pmatrix} * \begin{pmatrix} cs & -sn \\ sn & cs \end{pmatrix} = \begin{pmatrix} cs * y1 + sn * y2 & 0 \end{pmatrix}$$

with $cs^2 + sn^2 = 1$.

In other words, the action of `H` is equivalent to a standard Givens plane rotation, which zeros `y2`.

This subroutine is a square root free implementation of the two-way branch algorithm (fast plane rotations with dynamic scaling to avoid overflow/underflow) described in [Anda_Park:1994].

The arguments *X1* and *X2* are unchanged on return.

Synopsis:

```
call define_rot_fastgivens( x1 , x2 , d1 , d2 , beta , alpha , type_rot )
```

apply_rot_fastgivens()

purpose:

apply_rot_fastgivens() applies a fast Givens plane rotation H (defined by *BETA*, *ALPHA*, and *TYPE_ROT* on input) to the vector (y1 y2): or to the n-by-2 matrix [VECY1 VECY2]. That is,

$$\begin{pmatrix} y1 & y2 \end{pmatrix} * H = \begin{pmatrix} [h11 * y1 + h21 * y2] & [h12 * y1 + h22 * y2] \end{pmatrix}$$

or

$$[VECY1 \quad VECY2] * H = [(h11 * VECY1 + h21 * VECY2) \quad (h12 * VECY1 + h22 * VECY2)]$$

where H is a 2-by-2 matrix defined as

$$H = \begin{pmatrix} h11 & h12 \\ h21 & h22 \end{pmatrix}$$

More precisely, H takes one of the following forms:

- $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, if *TYPE_ROT* = 0.
- $\begin{pmatrix} 1 & 0 \\ B & 1 \end{pmatrix} \begin{pmatrix} 1 & A \\ 0 & 1 \end{pmatrix}$, if *TYPE_ROT* = 1
- $\begin{pmatrix} 1 & A \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ B & 1 \end{pmatrix}$, if *TYPE_ROT* = 2
- $\begin{pmatrix} 0 & -1 \\ 1 & A \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -B & 1 \end{pmatrix}$, if *TYPE_ROT* = 3
- $\begin{pmatrix} B & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & A \\ 0 & -1 \end{pmatrix}$, if *TYPE_ROT* = 4

Synopsis:

```
call apply_rot_fastgivens( y1,         y2,         beta, alpha, type_rot )
call apply_rot_fastgivens( vecy1(:n), vecy2(:n), beta, alpha, type_rot )
```

fastgivens_vec()

purpose:

fastgivens_vec() generates and applies a fast Givens plane rotation H to the n-by-2 matrix [VECX1 VECX2]. The rotation is designed to zero VECX2(1). That is,

$$[VECX1 \quad VECX2] * H = [(h11 * VECX1 + h21 * VECX2) \quad (h12 * VECX1 + h22 * VECX2)]$$

where $h12 * VECX1(1) + h22 * VECX2(1) = 0$ and H is the 2-by-2 matrix:

$$H = \begin{pmatrix} h11 & h12 \\ h21 & h22 \end{pmatrix}$$

Furthermore, the scale factors (*D1* and *D2*) are updated accordingly. That is, if on input:

$$[Y1 \quad Y2] = [\sqrt{d1} * VECX1 \quad \sqrt{d2} * VECX2]$$

then on output:

$$[\sqrt{d1} * VECX1 \quad \sqrt{d2} * VECX2] = [Y1 \quad Y2] * \begin{pmatrix} cs & -sn \\ sn & cs \end{pmatrix} =$$
$$[(cs * Y1 + sn * Y2) \quad (-sn * Y1 + cs * Y2)]$$

with $cs^2 + sn^2 = 1$ and $-sn * Y1(1) + cs * Y2(1) = 0$.

In other words, the action of H is equivalent to a standard Givens plane rotation, which zeros $Y2(1) = \sqrt{d2} * VECX2(1)$.

See the subroutine *define_rot_fastgivens()* for further details on the form of H.

Synopsis:

```
call fastgivens_vec( vecx1(:n) , vecx2(:n) , d1 , d2
↪)
call fastgivens_vec( vecx1(:n) , vecx2(:n) , d1 , d2, beta , alpha , type_rot
↪)
```

fastgivens_mat_left()

purpose:

fastgivens_mat_left() reduces the matrix MAT to upper trapezoidal form by applying a series of fast Givens plane rotations on the rows of MAT.

The (row) scale factors (MATD) are updated accordingly.

Synopsis:

```
call fastgivens_mat_left( mat(:p,:n), matd(:p) )
```

Examples:

ex1_fastgivens_mat_left.F90

fastgivens_mat_right()

purpose:

fastgivens_mat_right() reduces the matrix MAT to lower trapezoidal form by applying a series of fast Givens plane rotations on the columns of MAT.

The (column) scale factors (MATD) are updated accordingly.

Synopsis:

```
call fastgivens_mat_right( mat(:p,:n), matd(:n) )
```

Examples:

ex1_fastgivens_mat_right.F90

fastgivens_vec_mat_left()

purpose:

fastgivens_vec_mat_left() defines and applies a series of fast Givens plane rotations on the n-vector VEC and on the rows of a m-by-n matrix MAT. The rotations are designed to annihilate all the elements of the first column of MAT.

The (row) scale factors (VECD and MATD) are updated accordingly.

Synopsis:

```
call fastgivens_vec_mat_left( vec(:n) , mat(:p,:n), vecd , matd(:p) )
```

fastgivens_vec_mat_right()

purpose:

fastgivens_vec_mat_right() defines and applies a series of fast Givens plane rotations on the data:*m*-vector VEC and on the columns of a m-by-n matrix MAT. The rotations are designed to annihilate all the elements of the first row of MAT.

The (column) scale factors (VECD and MATD) are updated accordingly.

Synopsis:

```
call fastgivens_vec_mat_right( vec(:p) , mat(:p,:n) , vecd , matd(:n) )
```

define_rot_fastgivens2()

purpose:

define_rot_fastgivens2() generates a fast Givens plane rotation H (defined by *BETA*, *ALPHA*, and *TYPE_ROT* on output) and updated scale factors (*D1* and *D2*) , which zero *X2*. That is,

$$\begin{pmatrix} x1 & x2 \end{pmatrix} * H = \begin{pmatrix} r & 0 \end{pmatrix}$$

where H is equal to

- $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, if *TYPE_ROT* = 0.
- $\begin{pmatrix} 1 & 0 \\ B & 1 \end{pmatrix} \begin{pmatrix} 1 & A \\ 0 & 1 \end{pmatrix}$, if *TYPE_ROT* = 1
- $\begin{pmatrix} 1 & A \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ B & 1 \end{pmatrix}$, if *TYPE_ROT* = 2
- $\begin{pmatrix} 0 & -1 \\ 1 & A \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -B & 1 \end{pmatrix}$, if *TYPE_ROT* = 3
- $\begin{pmatrix} B & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & A \\ 0 & -1 \end{pmatrix}$, if *TYPE_ROT* = 4

On output, the arguments *BETA* = B and *ALPHA* = A and *TYPE_ROT* define the transformation matrix H:

$$H = \begin{pmatrix} h11 & h12 \\ h21 & h22 \end{pmatrix}$$

Furthermore, if on input, `y1 = x1*d1` and `y2 = x2*d2`, then on output, with the updated scale factors *D1* and *D2*:

$$\begin{pmatrix} x1 & x2 \end{pmatrix} * H * diag(d1 \quad d2) = \begin{pmatrix} [x1 * h11 + x2 * h21] * d1 & 0 \end{pmatrix}$$

is equal to

$$\begin{pmatrix} y1 & y2 \end{pmatrix} * \begin{pmatrix} cs & -sn \\ sn & cs \end{pmatrix} = \begin{pmatrix} cs * y1 + sn * y2 & 0 \end{pmatrix}$$

with $cs^2 + sn^2 = 1$.

In other words, the action of H is equivalent to a standard Givens plane rotation, which zeros y2.

This subroutine is an implementation of the two-way branch algorithm (fast plane rotations with dynamic scaling to avoid overflow/underflow) described in [Anda_Park:1994].

The arguments *X1* and *X2* are unchanged on return.

Synopsis:

```
call define_rot_fastgivens2( x1 , x2 , d1 , d2 , beta , alpha , type_rot )
```

fastgivens2_vec()

purpose:

fastgivens2_vec() generates and applies a fast Givens plane rotation H to the n-by-2 matrix `[VECX1 VECX2]`. The rotation is designed to zero `VECX2(1)`. That is,

$$[VECX1 \quad VECX2] * H = [(h11 * VECX1 + h21 * VECX2) \quad (h12 * VECX1 + h22 * VECX2)]$$

where $h12 * VECX1(1) + h22 * VECX2(1) = 0$ and H is the 2-by-2 matrix:

$$H = \begin{pmatrix} h11 & h12 \\ h21 & h22 \end{pmatrix}$$

Furthermore, the scale factors (*D1* and *D2*) are updated accordingly. That is, if on input:

$$[Y1 \quad Y2] = [d1 * VECX1 \quad d2 * VECX2]$$

then on output:

$$[d1 * VECX1 \quad d2 * VECX2] = [Y1 \quad Y2] * \begin{pmatrix} cs & -sn \\ sn & cs \end{pmatrix} =$$

$$[(cs * Y1 + sn * Y2) \quad (-sn * Y1 + cs * Y2)]$$

with $cs^2 + sn^2 = 1$ and $-sn * Y1(1) + cs * Y2(1) = 0$.

In other words, the action of `H` is equivalent to a standard Givens plane rotation, which zeros $Y2(1) = d2 * VECX2(1)$.

See the subroutine *define_rot_fastgivens2()* for further details on the form of `H`.

Synopsis:

```
call fastgivens2_vec( vecx1(:n) , vecx2(:n) , d1 , d2
↪ )
call fastgivens2_vec( vecx1(:n) , vecx2(:n) , d1 , d2, beta , alpha , type_
↪rot )
```

5.17 MODULE Hous_Procedures

Module *Hous_Procedures* exports subroutines for computing and applying elementary Householder reflectors [Golub_VanLoan:1996] [Lawson_Hanson:1974].

A Householder transformation is a rank-`1` modification of the identity matrix which can be used to zero out selected elements of a vector. A `n`-by-`n` Householder reflector matrix `H` in STATPACK is represented in the form

$$H = I + \tau * (v * v^T)$$

where `v` is a `n`-vector, called the Householder vector, and τ is a scalar. Note that a Householder reflector matrix `H` always verifies

$$H^T * H = I$$

and

$$H = H^T$$

The routines in the *Hous_Procedures* take into account the rank-`1` structure and the orthogonal and symmetry properties of Householder reflectors to create and apply Householder transformations efficiently.

Two different implementations of Householder reflectors are provided here, the first is described in [Anderson_Fahey:1997] and the second in [Lawson_Hanson:1974]. In both cases, improvements suggested in [Anderson:2018] and [Hanson_Hopkins:2018] for computing safely and accurately the `2`-norm of a vector have also been incorporated.

Please note, finally, that routines provided in this module apply only to real data of kind **stnd**. The real kind type **stnd** is defined in module *Select_Parameters*.

In order to use one of these routines, you must include an appropriate `use Hous_Procedures` or `use Statpack` statement in your Fortran program, like:

```
use Hous_Procedures, only: hous1
```

or :

```
use Statpack, only: hous1
```

Here is the list of the public routines exported by module *Hous_Procedures*:

hous1()

purpose:

Given a n-element real vector x (provided in the argument *U*), **hous1()** generates a real elementary reflector H of order n, such that

$$H * x = \begin{pmatrix} beta \\ 0 \end{pmatrix}$$

where beta is a real scalar. H is represented in the form

$$H = I + tau * (v * v^T)$$

where tau is a real scalar and v is a n-element real vector with v(1) = 1.

If the elements of x(2:n) are all zero or size(U)=1, then tau = 0 and H is taken to be the unit matrix.

Otherwise 1 <= tau <= 2.

This subroutine is based on the routine DLARFG() in LAPACK with improvements suggested by [Anderson_Fahey:1997] [Anderson:2018] and [Hanson_Hopkins:2018].

Synopsis:

```
call hous1( u(:n) , tau         )
call hous1( u(:n) , tau , beta )
```

Examples:

ex1_hous1.F90

apply_hous1()

purpose:

apply_hous1() applies a real elementary reflector H generated by *hous1()* to a real vector/matrix C. H is represented in the form

$$H = I + tau * (v * v^T)$$

where tau is a real scalar and v is a n-element real vector with v(1) = 1.

If tau = 0, then H is taken to be the unit matrix and C is not modified.

Synopsis:

```
call apply_hous1( u(:n) , tau , vec(:)          )
call apply_hous1( u(:n) , tau , mat(:,:) , left )
```

Examples:

ex1_hous1.F90

hous2()

purpose:

Given a `n-1`-element real vector `x` and a real scalar `alpha` (provided in the arguments *U* and *PIVOT* on entry), **hous2()** generates a real elementary reflector `H` of order `n`, such that

$$H * \begin{pmatrix} alpha \\ x \end{pmatrix} = \begin{pmatrix} beta \\ 0 \end{pmatrix}$$

where `beta` is a real scalar. `H` is represented in the form

$$H = I + tau * (v * v^T)$$

where `tau` is a real scalar and `v` is a `n`-element real vector with `v(1) = 1`.

If the elements of `x` are all zero, then `tau = 0` and `H` is taken to be the unit matrix.

Otherwise `1 <= tau <= 2`.

This subroutine is based on the routine DLARFG() in LAPACK with improvements suggested by [Anderson_Fahey:1997] [Anderson:2018] and [Hanson_Hopkins:2018].

Synopsis:

```
call hous2( pivot, u(:n-1) , tau  )
```

Examples:

ex1_hous2.F90

apply_hous2()

purpose:

apply_hous2() applies a real elementary reflector `H` of order `n`, generated by *hous2()*, to a real vector/matrix `C`. `H` is represented in the form

$$H = I + tau * (v * v^T)$$

where `tau` is a real scalar and `v` is a `n`-element real vector with `v(1) = 1`. More precisely, `v` is defined as

$$v = \begin{pmatrix} 1 \\ u \end{pmatrix}$$

for the input vector argument *U*.

Here, the `n`-element real vector `C` has the form:

$$C = \begin{pmatrix} piv \\ vec \end{pmatrix}$$

or the real matrix `C` has the form, if `LEFT=true`:

$$C = \begin{bmatrix} vec_piv^T \\ MAT \end{bmatrix}$$

or, if `LEFT=false`:

$$C = \begin{bmatrix} vec_piv & MAT \end{bmatrix}$$

for given input arguments *PIV*, *VEC* or *VEC_PIV*, *MAT*.

If `tau = 0`, then `H` is taken to be the unit matrix and `C` is not modified.

Synopsis:

```
call apply_hous2( u(:n-1) , tau ,  piv         , vec(:)           )
call apply_hous2( u(:n-1) , tau ,  vec_piv(:) , mat(:,:) , left )
```

Examples:

ex1_hous2.F90

h1 ()

purpose:

Given a n-element real vector x (provided in the argument *U*), **h1()** generates a real elementary reflector H of order n, such that

$$H * x = \begin{pmatrix} beta \\ 0 \end{pmatrix}$$

where beta is a real scalar. H is represented in the form

$$H = I + tau * (v * v^T)$$

where tau is a real scalar and v is a n-element real vector. Note that here, v(1) is not equal to 1, contrary to the formulation used in *hous1()*.

The real elementary reflector H is then, optionally, applied to a real vector/matrix C.

This subroutine is based on the routine H1 described in [Lawson_Hanson:1974] with improvements suggested in [Anderson:2018] and [Hanson_Hopkins:2018].

Synopsis:

```
call h1( u(:n) , beta , tau                    )
call h1( u(:n) , beta , tau , vec(:)           )
call h1( u(:n) , beta , tau , mat(:,:) , left )
```

Examples:

ex1_h1.F90

apply_h1 ()

purpose:

apply_h1() applies a real elementary reflector H generated by *h1()* to a n-element real vector or to a n-by-m or m-by-n real matrix from the left or the right. H is represented in the form

$$H = I + tau * (v * v^T)$$

where tau is a real scalar and v is a n-element real vector.

Synopsis:

```
call apply_h1( u(:n) , tau , vec(:)           )
call apply_h1( u(:n) , tau , mat(:,:) , left )
```

Examples:

ex1_h1.F90

h2 ()

purpose:

Given a n-1-element real vector x and a real scalar alpha (provided in the arguments *U* and *BETA* on entry), **h2()** generates a real elementary reflector H of order n, such that

$$H * \begin{pmatrix} alpha \\ x \end{pmatrix} = \begin{pmatrix} beta \\ 0 \end{pmatrix}$$

where beta is a real scalar. H is represented in the form

$$H = I + tau * (v * v^T)$$

where `tau` is a real scalar and `v` is a `n`-element real vector. Note that here `v(1)` is not equal to `1`, contrary to the formulation in *hous2()*.

The real elementary reflector `H` is then, optionally, applied to a real vector/matrix `C`.

This subroutine is based on the routine H2 described in [Lawson_Hanson:1974] with improvements suggested in [Anderson:2018] and [Hanson_Hopkins:2018].

Synopsis:

```
call h2( beta, u(:n-1) , up , tau                              )
call h2( beta, u(:n-1) , up , tau , piv         , vec(:)       )
call h2( beta, u(:n-1) , up , tau , vec_piv(:) , mat(:) , left )
```

Examples:

ex1_h2.F90

apply_h2()

purpose:

apply_h2() applies a real elementary reflector `H` generated by *h2()* to a `n`-element real vector or to a `n`-by-`m` or `m`-by-`n` real matrix from the left or the right. `H` is represented in the form

$$H = I + tau * (v * v^T)$$

where `tau` is a real scalar and `v` is a `n`-element real vector. More precisely, `v` is defined as

$$v = \begin{pmatrix} up \\ u \end{pmatrix}$$

from the input arguments *UP* and *U*.

Here, the `n`-element real vector `C` has the form:

$$C = \begin{pmatrix} piv \\ vec \end{pmatrix}$$

or the real matrix `C` has the form, if `LEFT=true`:

$$C = \begin{bmatrix} vec_piv^T \\ MAT \end{bmatrix}$$

or, if `LEFT=false`:

$$C = \begin{bmatrix} vec_piv & MAT \end{bmatrix}$$

for given input arguments *PIV*, *VEC* or *VEC_PIV*, *MAT*.

If `tau` = `0`, then `H` is taken to be the unit matrix and `C` is not modified.

Synopsis:

```
call apply_h2( u(:n-1) , up , tau ,  piv         , vec(:)          )
call apply_h2( u(:n-1) , up , tau ,  vec_piv(:) , mat(:,:) , left )
```

Examples:

ex1_h2.F90

5.18 MODULE QR_Procedures

Module *QR_Procedures* exports subroutines for computing (full or partial) QR and LQ decompositions and related factorizations or computations.

A general rectangular m-by-n matrix MAT has a QR decomposition into the product of an orthogonal m-by-m square matrix Q (where $Q^T * Q = I$) and a m-by-n upper-triangular (or upper trapezoidal) matrix R [Lawson_Hanson:1974] [Golub_VanLoan:1996] [Hansen_etal:2012],

$$MAT = Q * R$$

Similarly, MAT has a LQ decomposition into the product of a m-by-n lower-triangular (or lower trapezoidal) matrix L and an orthogonal n-by-n square matrix Q (where $Q^T * Q = I$) [Lawson_Hanson:1974] [Golub_VanLoan:1996],

$$MAT = L * Q$$

The QR decomposition can be used to convert a full rank n-by-n linear system $MAT * x = b$ into the triangular system $R * x = Q^T * b$, which can be solved by back-substitution.

Similarly, the LQ decomposition can be used to convert a full rank n-by-n linear system $x * MAT = b$ into the triangular system $x * L = b * Q^T$, which can also be solved by back-substitution.

The QR or LQ decompositions can also be used to solve linear least squares problems, when MAT has full rank [Lawson_Hanson:1974] [Golub_VanLoan:1996] [Hansen_etal:2012]. See the module *LLSQ_Procedures* for more details.

Another use of the QR or LQ decompositions is to compute an orthonormal basis for a set of vectors. The first min(m,n) columns of Q of the QR decomposition form an orthonormal basis for the range of MAT, $\mathrm{ran}(MAT)$, when MAT has full rank. Similarly, the first min(m,n) rows of Q of the LQ decomposition form an orthonormal basis for the range of $\mathtt{MAT}^{\mathtt{T}}$, when MAT has full rank.

The QR decomposition of a m-by-n matrix MAT can be extended to the rank deficient case by introducing a column permutation P [Lawson_Hanson:1974] [Golub_VanLoan:1996] [Hansen_etal:2012],

$$MAT * P = Q * R = Q * \begin{bmatrix} R11 & R12 \\ 0 & R22 \end{bmatrix} \simeq \begin{bmatrix} R11 & R12 \\ 0 & 0 \end{bmatrix}$$

where P is a permutation of the columns of $\mathtt{I}_\mathtt{n}$, the identity matrix of order *n*, R11 is a r-by-r full rank upper triangular matrix, R12 is a r-by-n−r matrix and R22 is a (m−r)-by-(n−r) upper triangular matrix, which is almost negligible. In other words, when MAT is rank deficient with $r = \mathrm{rank}(MAT)$, the matrix R can be partitioned into four submatrices and the dimension of R11 is equal to $\mathrm{rank}(MAT)$. The effective rank of MAT, r, can be estimated by the routines provided here.

When MAT is square and of full rank (e.g., r = m = n), this decomposition can also be used to convert the linear system $MAT * x = b$ into the triangular system $R * y = Q^T * b$, $x = P * y$, which can be solved by back-substitution and permutation.

More generally, for a matrix with column rank r, The first r columns of Q form an orthonormal basis for the range of MAT and the QR decomposition with Column Pivoting (QRCP) can be used to solve rank deficient linear least squares problems. See the manual of the module *LLSQ_Procedures* for more details.

Finally, the Complete Orthogonal Decomposition (COD) of a m-by-n matrix MAT [Lawson_Hanson:1974] [Golub_VanLoan:1996] [Hansen_etal:2012] is a generalization of the QRCP described above, given by

$$MAT * P = Q * T * Z$$

where P is a n-by-n permutation matrix, Q is a m-by-m orthogonal matrix, Z is a n-by-n orthogonal matrix and T is a m-by-n matrix.

If MAT has full column rank, then T = R, Z = I and this reduces to the QRCP. On the other hand, if MAT is column deficient, T has the form:

$$T = \begin{bmatrix} T11 & 0 \\ 0 & 0 \end{bmatrix}$$

where T11 is a r-by-r upper triangular full rank matrix and r is the effective rank of MAT.

The advantage of using the COD for rank deficient matrices is the ability to compute the minimum norm solution to the linear least squares problem $\min_x ||b - MAT * x||_2$. See description of the routines in module *LLSQ_Procedures* for more details.

All these different matrix decompositions can be performed with routines available in this module. Moreover, most routines in this module are blocked and multi-threaded versions [Walker:1988] [Dongarra_etal:1989] of the standard sequential algorithms for the QR, LQ, QRCP and COD decompositions [Lawson_Hanson:1974] [Golub_VanLoan:1996]. All routines available in module *QR_Procedures* are deterministic, but randomized full or partial QRCP and COD decompositions [Duersch_Gu:2017] [Martinsson_etal:2017] [Duersch_Gu:2020] [Martinsson:2019], which perform the same tasks and are much faster than their deterministic versions, are also available in module *Random* and can be used efficiently for very large matrices.

Please note, finally, that routines provided in this module apply only to real data types.

In order to use one of these routines, you must include an appropriate `use QR_Procedures` or `use Statpack` statement in your Fortran program, like:

```
use QR_Procedures, only: lq_cmp
```

or :

```
use Statpack, only: lq_cmp
```

Here is the list of the public routines exported by module *QR_Procedures*:

lq_cmp()

Purpose:

lq_cmp() computes a LQ factorization of a real m-by-n matrix MAT :

$$MAT = L * Q$$

where Q is orthogonal and L is lower trapezoidal (lower triangular if m<=n).

Synopsis:

```
call lq_cmp( mat(:m,:n) , diagl(:min(m,n)) , tau(:min(m,n)) , use_qr=use_qr )
```

Examples:

ex1_lq_cmp.F90

ortho_gen_lq()

Purpose:

ortho_gen_lq() generates an m-by-n real matrix with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$$Q = H(k) * ... * H(2) * H(1)$$

as returned by *lq_cmp()*.

Synopsis:

```
call ortho_gen_lq( mat(:m,:n) , tau(:p) , use_qr=use_qr )
```

Examples:

ex1_lq_cmp.F90

apply_q_lq()

Purpose:

apply_q_lq() overwrites the general real m-by-n matrix C with

- $Q * C$ if *LEFT* = true and *TRANS* = false,
- $Q^T * C$ if *LEFT* = true and *TRANS* = true,
- $C * Q$ if *LEFT* = false and *TRANS* = false,
- $C * Q^T$ if *LEFT* = false and *TRANS* = true,

or the real n-vector C with

- $Q * C$ if *TRANS* = false,
- $Q^T * C$ if *TRANS* = true,

where Q is a real orthogonal matrix defined as the product of k elementary reflectors

$$Q = H(k) * ... * H(2) * H(1)$$

as returned by *lq_cmp()*. Q is of order m if *LEFT* = true and of order n if *LEFT* = false when C is a matrix.

Synopsis:

```
call apply_q_lq( mat(:m,:n) , tau(:p) , c(:n)       ,         trans
↪     )
call apply_q_lq( mat(:m,:n) , tau(:p) , c(:mc,:nc) , left , trans , use_
↪qr=use_qr )
```

qr_cmp()

Purpose:

qr_cmp() computes a QR factorization of a real m-by-n matrix MAT :

$$MAT = Q * R$$

where Q is orthogonal and R is upper trapezoidal (upper triangular if m>=n).

Synopsis:

```
call qr_cmp( mat(:m,:n) , diagr(:p) , beta(:p) )
```

Examples:

ex1_qr_cmp.F90

ex2_qr_cmp.F90

ex1_qr_solve.F90

ex1_random_svd.F90

ex1_random_eig.F90

ex1_random_eig_pos.F90

qr_cmp_ts()

Purpose:

qr_cmp_ts() computes a QR factorization of a real m-by-n matrix MAT :

$$MAT = Q * R$$

where Q is orthogonal and R is upper trapezoidal (upper triangular if m>=n).

qr_cmp_ts() performs exactly the same task as *qr_cmp()* and is usually slower than *qr_cmp()*, but may be more robust for huge tall and skinny matrices.

Synopsis:

```
call qr_cmp_ts( mat(:m,:n) , diagr(:p) , beta(:p) )
```

Examples:

ex1_qr_cmp_ts.F90

ex2_qr_cmp_ts.F90

qr_cmp2 ()

Purpose:

qr_cmp2() computes a (complete) orthogonal factorization of a real m-by-n matrix MAT. MAT may be rank-deficient. The routine first computes a QRCP of MAT:

$$MAT * P = Q * R$$

here P is n-by-n permutation matrix, R is an upper triangular or trapezoidal (if n>m) matrix and Q is a m-by-m orthogonal matrix.

R can then be partitioned by defining R11 as the largest leading squared submatrix of R whose estimated condition number, in the 1-norm, is less than 1/*TOL* (or such that $|R(j,j)| > 0$ if the argument *TOL* is absent). The order of R11, krank, is the effective rank of MAT.

This leads to the following partition of R:

$$R = \begin{bmatrix} R11 & R12 \\ 0 & R22 \end{bmatrix} \simeq \begin{bmatrix} R11 & R12 \\ 0 & 0 \end{bmatrix}$$

where R22 can be considered to be negligible.

If *TAU* is present, R22 is considered to be negligible and R12 is annihilated by orthogonal transformations from the right, arriving at the complete orthogonal factorization:

$$MAT * P \simeq Q * \begin{bmatrix} T11 & 0 \\ 0 & 0 \end{bmatrix} * Z$$

where P is a n-by-n permutation matrix, Q is a m-by-m orthogonal matrix, Z is a n-by-n orthogonal matrix and T11 is a krank-by-krank upper triangular matrix.

Note, finally, that randomized versions of subroutine **qr_cmp2()** are available in module *Random*, which perform the same tasks with nearly the same accuracy and are much faster. **qr_cmp2()** uses a standard "BLAS2" algorithm without any blocking and is thus not optimized for computing QRCP or COD factorizations of very large matrices. For large matrices, subroutines *partial_rqr_cmp()* and *partial_rqr_cmp2()* in module *Random*, which use randomized blocked "BLAS3" algorithms described in [Duersch_Gu:2017] [Martinsson_etal:2017] [Xiao_etal:2017] [Duersch_Gu:2020], are a much better choice.

Synopsis:

```
call qr_cmp2( mat(:m,:n) , diagr(:min(m,n)) , beta(:min(m,n)) , ip(:n) ,␣
↪krank , tol=tol , tau=tau(:min(m,n)) )
```

Examples:

ex1_qr_cmp2.F90

ex2_qr_cmp2.F90

ex3_qr_cmp2.F90

ex1_qr_solve2.F90

partial_qr_cmp()

Purpose:

partial_qr_cmp() computes a (partial) QRCP or COD of a real m-by-n matrix MAT. MAT may be rank-deficient. In other words, **partial_qr_cmp()** performs the same task as *qr_cmp2()* above, but allows to stop the factorization at any stage in order to obtain only a partial QRCP or COD factorization of MAT. This option is not allowed with *qr_cmp2()*, which always computes a full QRCP in its first stage.

The routine first computes a QRCP of MAT:

$$MAT * P = Q * R$$

here P is n-by-n permutation matrix, R is an upper triangular or trapezoidal (if n>m) matrix and Q is a m-by-m orthogonal matrix.

At the user option, the QR factorization can be only partial, e.g., the subroutine ends when the numbers of selected columns of MAT for pivot selection is equal to a predefined value equals to kpartial = size(DIAGR) = size(BETA).

This leads implicitly to the following partition of Q:

$$Q = \begin{bmatrix} Q1 & Q2 \end{bmatrix}$$

where Q1 is a m-by-kpartial orthonormal matrix and Q2 is a m-by-(m−kpartial) orthonormal matrix orthogonal to Q1, and to the following corresponding partition of R:

$$R = \begin{bmatrix} R11 & R12 \\ 0 & R22 \end{bmatrix}$$

where R11 is a kpartial-by-kpartial triangular matrix, R21 is zero by construction, R12 is a full kpartial-by-(n−kpartial) matrix and R22 is a full (m−kpartial)-by-(n−kpartial) matrix.

From these partitions of Q and R, we can obtain a good approximation of MAT of rank kpartial, since:

$$MAT * P \simeq Q1 * \begin{bmatrix} R11 & R12 \end{bmatrix}$$

and, finally:

$$MAT \simeq Q1 * \begin{bmatrix} R11 & R12 \end{bmatrix} * P^T$$

which is equivalent to assume that R22 is negligible.

If *TAU* is present, R12 is then annihilated by orthogonal transformations from the right, arriving at the partial orthogonal factorization:

$$MAT * P \simeq Q1 * \begin{bmatrix} T11 & 0 \end{bmatrix} * Z$$

where P is a n-by-n permutation matrix, Q1 is a m-by-kpartial orthonormal matrix, Z is a n-by-n orthogonal matrix and T11 is a kpartial-by-kpartial upper triangular matrix.

As in subroutine *qr_cmp2()*, if the optional argument *TOL* is present, calculations to determine the 1-norm condition number of R11 are performed and this condition number is used to determine the effective pseudo-rank of R11, krank. If this effective pseudo-rank is less than kpartial, which implies that the rank of MAT is also less than kpartial, the subroutine outputs a partial QR factorization corresponding to this effective pseudo-rank krank, instead of rank kpartial.

In all cases, the subroutine outputs krank (or kpartial if *TOL* is absent) in the argument *KRANK* and $||MAT(krank+1:m, krank+1:n)||_F$ gives the error of the associated matrix approximation in the Frobenius norm, on exit.

Note, finally, that randomized versions of subroutine **partial_qr_cmp()** are available in module *Random*, which perform the same tasks with nearly the same accuracy and are much faster. **partial_qr_cmp()** uses a standard "BLAS2"

algorithm without any blocking and is thus not optimized for computing a partial QRCP or COD of very large matrices. For large matrices, subroutines *partial_rqr_cmp()* and *partial_rqr_cmp2()* in module *Random*, which use randomized blocked "BLAS3" algorithms described in [Duersch_Gu:2017] [Martinsson_etal:2017] [Xiao_etal:2017] [Duersch_Gu:2020], are a much better choice.

Synopsis:

```
call partial_qr_cmp( mat(:m,:n) , diagr(:kpartial) , beta(:kpartial) , ip(:n)␣
↪, krank , tol=tol , tau=tau(:kpartial) )
```

Examples:

ex1_partial_qr_cmp.F90

ex2_partial_qr_cmp.F90

ex3_partial_qr_cmp.F90

ex1_cur_cmp.F90

partial_qr_cmp_fixed_precision()

Purpose:

partial_qr_cmp_fixed_precision() computes a partial QRCP (or COD) of a real m-by-n matrix MAT:

$$MAT * P \simeq Q * R$$

here P is a n-by-n permutation matrix, R is a kank-by-n upper triangular or trapezoidal matrix and Q is a m-by-krank with orthonormal columns. This leads to the following matrix approximation of MAT of rank krank:

$$MAT \simeq Q * R * P^T$$

krank is the target rank of the matrix approximation, which is sought, and this partial factorization must have an approximation error which fulfills:

$$||MAT - Q * R * P^T||_F <= ||MAT||_F * relerr$$

where $||||_F$ is the Frobenius norm and relerr is a prescribed accuracy tolerance for the relative error of the computed matrix approximation, specified in the input argument *RELERR*.

Thus, **partial_qr_cmp_fixed_precision**() performs exactly the same task as *partial_qr_cmp()* above, but allows to stop the factorization at any stage in order to obtain a partial QR (or orthogonal) factorization of MAT, which fullfills the above inequality.

In other words, krank, the rank of the matrix approximation, is not known in advance and is computed by the subroutine. Otherwise, all other arguments of **partial_qr_cmp_fixed_precision**() have the same meaning as in *qr_cmp2()* or *partial_qr_cmp()*.

In all cases, on exit, $||MAT(krank+1:m, krank+1:n)||_F$ gives the error of the associated matrix approximation in the Frobenius norm and the associated relative error in the Frobenius norm is output in argument *RELERR*.

Note, finally, that a randomized version of subroutine **partial_qr_cmp_fixed_precision**() is available in module *Random*, which performs the same tasks with nearly the same accuracy and is much faster. **partial_qr_cmp_fixed_precision**() uses a standard "BLAS2" algorithm without any blocking and is thus not optimized for computing partial QRCPs or CODs of very large matrices. For large matrices, subroutine *partial_rqr_cmp_fixed_precision()* in module *Random*, which uses a randomized blocked "BLAS3" algorithm described in [Duersch_Gu:2017] [Martinsson_etal:2017] [Xiao_etal:2017] [Duersch_Gu:2020], is a much better choice.

Synopsis:

```
call partial_qr_cmp_fixed_precision( mat(:m,:n) , relerr , diagr(:min(m,n)) ,␣
↪beta(:min(m,n)) , ip(:n) , krank , tau=tau(:min(m,n)) )
```

Examples:

ex1_partial_qr_cmp_fixed_precision.F90

qrfac()

Purpose:

qrfac() is a low level subroutine for computing a (complete) orthogonal factorization of the array section `SYST(1:m, 1:n)` where `n` <= `size(SYST,2)` and `m` = `size(SYST,1)`.

The routine first computes a QRCP:

$$SYST(1:m, 1:n) * P = Q * R$$

where `P` is `n`-by-`n` permutation matrix, `R` is an upper triangular or trapezoidal (if `n>m`) matrix and `Q` is a `m`-by-`m` orthogonal matrix.

The orthogonal transformation `Q` is then applied to `SYST(1:m,n+1:)`:

$$SYST(1:m, n+1:) = Q * B$$

Then, the rank of `SYST(1:m,1:n)` is determined by finding the squared submatrix `R11` of `R` which is defined as the largest leading submatrix whose estimated condition number, in the `1`-norm, is less than `1`/*TOL* or such that $|R(j,j)| > 0$ if *TOL* is absent. The order of `R11`, `krank`, is the effective rank of `SYST(1:m,1:n)`.

This leads to the following partition of `R`:

$$R = \begin{bmatrix} R11 & R12 \\ 0 & R22 \end{bmatrix} \simeq \begin{bmatrix} R11 & R12 \\ 0 & 0 \end{bmatrix}$$

where `R22` can be considered to be negligible.

If *MIN_NORM* = `true`, `R22` is considered to be negligible and `R12` is annihilated by orthogonal transformations from the right, arriving at the complete orthogonal factorization:

$$SYST(1:m, 1:n) * P \simeq Q * \begin{bmatrix} T11 & 0 \\ 0 & 0 \end{bmatrix} * Z$$

where `P` is a `n`-by-`n` permutation matrix, `Q` is a `m`-by-`m` orthogonal matrix, `Z` is a `n`-by-`n` orthogonal matrix and `T11` is a `krank`-by-`krank` upper triangular matrix.

Synopsis:

```
call qrfac( name_proc , syst(:m,:n) , kfix , krank , min_norm, diagr(:) ,␣
↪beta(:) , h(:) , tol=tol , ip=ip(:) )
```

ortho_gen_qr()

Purpose:

ortho_gen_qr() generates an `m`-by-`n` real matrix with orthonormal columns, which is defined as the first `n` columns of a product of `k` elementary reflectors of order `m`

$$Q = H(1) * H(2) * ... * H(k)$$

as returned by *qr_cmp()*, *qr_cmp_ts()* or *qr_cmp2()*.

Synopsis:

```
call ortho_gen_qr( mat(:m,:n) , beta(:p) )
```

Examples:

ex1_qr_cmp.F90

ex1_qr_cmp2.F90

ex1_partial_qr_cmp.F90

ex1_partial_rqr_cmp.F90

ex1_partial_rqr_cmp2.F90

ex1_partial_rtqr_cmp.F90

ex1_random_svd.F90

ex1_random_eig.F90

ex1_random_eig_pos.F90

ortho_gen_qr_ts()

Purpose:

ortho_gen_qr_ts() generates an m-by-n real matrix with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m

$$Q = H(1) * H(2) * ... * H(k)$$

as returned by *qr_cmp()*, *qr_cmp_ts()* or *qr_cmp2()*.

ortho_gen_qr_ts() performs exactly the same task as *ortho_gen_qr()* and is usually slower than *ortho_gen_qr()*, but may be more robust for huge tall and skinny matrices.

Synopsis:

```
call ortho_gen_qr_ts( mat(:m,:n) , beta(:p) )
```

Examples:

ex1_qr_cmp_ts.F90

apply_q_qr()

Purpose:

apply_q_qr() overwrites the general real m-by-n matrix C with

- $Q * C$ if *LEFT* = true and *TRANS* = false,
- $Q^T * C$ if *LEFT* = true and *TRANS* = true,
- $C * Q$ if *LEFT* = false and *TRANS* = false,
- $C * Q^T$ if *LEFT* = false and *TRANS* = true,

or the real m-vector C with

- $Q * C$ if *TRANS* = false,
- $Q^T * C$ if *TRANS* = true,

where Q is a real orthogonal matrix defined as the product of k elementary reflectors

$$Q = H(1) * H(2) * ... * H(k)$$

as returned by *qr_cmp()*, *qr_cmp_ts()* or *qr_cmp2()*. Q is of order m if *LEFT* = true and of order n if *LEFT* = false when C is a matrix.

Synopsis:

```
call apply_q_qr( mat(:m,:n) , beta(:p) , c(:m)      ,        trans )
call apply_q_qr( mat(:m,:n) , beta(:p) , c(:mc,:nc) , left , trans )
```

Examples:

ex2_bd_inviter.F90

apply_q_qr_ts()

Purpose:

apply_q_qr_ts() overwrites the general real m-by-n matrix C with

- $Q * C$ if *LEFT* = true and *TRANS* = false,
- $Q^T * C$ if *LEFT* = true and *TRANS* = true,
- $C * Q$ if *LEFT* = false and *TRANS* = false,
- $C * Q^T$ if *LEFT* = false and *TRANS* = true,

or the real m-vector C with

- $Q * C$ if *TRANS* = false,
- $Q^T * C$ if *TRANS* = true,

where Q is a real orthogonal matrix defined as the product of k elementary reflectors

$$Q = H(1) * H(2) * ... * H(k)$$

as returned by *qr_cmp()*, *qr_cmp_ts()* or *qr_cmp2()*. Q is of order m if *LEFT* = true and of order n if *LEFT* = false when C is a matrix.

apply_q_qr_ts() performs exactly the same task as *apply_q_qr()* and is usually slower than *apply_q_qr()*, but may be more robust for huge tall and skinny matrices.

Synopsis:

```
call apply_q_qr_ts( mat(:m,:n) , beta(:p) , c(:m)      ,         trans )
call apply_q_qr_ts( mat(:m,:n) , beta(:p) , c(:mc,:nc) , left , trans )
```

5.19 MODULE Eig_Procedures

Module *EIG_Procedures* exports a large set of procedures for computing (selected) eigenvalues and/or (selected) eigenvectors of a symmetric (tridiagonal) matrix [Lawson_Hanson:1974] [Golub_VanLoan:1996] [Parlett:1998]. Fast methods for obtaining approximations of a truncated EigenValue Decomposition (EVD) of symmetric matrices based on recent randomization algorithms are also included [Halko_etal:2011] [Martinsson:2019].

The standard real symmetric eigenvalue problem is to find eigenvalues λ and eigenvectors u such that

$$MAT * u = \lambda * u$$

where MAT is a n-by-n real symmetric matrix.

For an input n-by-n dense matrix MAT, this module provides routines for:

- the transformation of MAT to tridiagonal form T,

  $$Q^T * MAT * Q = T$$

  where Q is a n-by-n orthogonal matrix;

- the computation of the eigenvalues λ_i and eigenvectors p_i of the tridiagonal matrix T,

  $$P^T * T * P = S$$

  where S is a n-by-n diagonal matrix with $S(i,i) = \lambda_i$ and P is the n-by-n matrix of associated eigenvectors of T;

- the back-transformation of the eigenvectors p_i of T to eigenvectors u_i of MAT,

$$MAT = (Q * P) * S * (Q * P)^T = U * S * U^T$$

  where U is the n-by-n matrix of eigenvectors of MAT and the eigenvalues $S(i,i) = \lambda_i$ of T are also the eigenvalues of MAT.

The transformation of MAT to tridiagonal form T, the generation of the associated orthogonal matrix Q, the computation of the eigenvectors of T and the back-transformation the eigenvectors of T to eigenvectors of MAT can be done with multi-threaded and blocked algorithms at the user option [Dongarra_etal:1989] [Golub_VanLoan:1996] [Walker:1988]. All algorithms are parallelized with OpenMP [openmp]. Depending on the situation and the algorithm used, it is also possible to compute selected, or only the largest or smallest eigenvalues and the associated eigenvectors.

Currently, STATPACK includes four different algorithms for computing (selected) eigenvalues of a tridiagonal matrix T:

- the implicit QR method [Lawson_Hanson:1974] [Golub_VanLoan:1996] [Parlett:1998] ;
- the fast Pal-Walker-Kahan variant of the implicit QR method, which does not use square roots [Parlett:1998] ;
- the bisection method, which is based on Sturm sequences and requires $O(n.k)$ operations to compute k eigenvalues of T [Golub_VanLoan:1996] [Parlett:1998] ;
- and a rational QR algorithm, where the shift is determined using Newton's method and makes it possible to *steer* the iterations toward desired eigenvalues [Reinsch_Bauer:1968]. This method also allows computation of subset of eigenvalues as the bisection method.

These STATPACK eigenvalues routines generally compute eigenvalues of T (or MAT) to an absolute accuracy of $\epsilon||T||_2$ (or $\epsilon||MAT||_2$), where ϵ is the machine precision and $||||_2$ is the spectral norm. If higher accuracy is wanted, subroutine *symtrid_bisect()* (or the drivers *select_eigval_cmp3()* or *eig_cmp5()* in case of a dense matrix MAT), which uses the bisection method, should be used with the optional argument *ABSTOL* set to sqrt(lamch("S")) (which is equal to the public numerical constant safmin in the *Num_Constants* module).

Currently, STATPACK includes three different methods for computing eigenvectors of a symmetric tridiagonal matrix T:

- implicit QR iterations [Golub_VanLoan:1996] [Parlett:1998];
- inverse iteration combined with Fernando's method or random starting vectors [Golub_VanLoan:1996] [Ipsen:1997] [Dhillon:1998] [Fernando:1997] [Bini_etal:2005];
- and a deflation algorithm inspired from the work of Godunov and collaborators, which is also related to Fernando's method for computing eigenvectors [Godunov_etal:1993] [Malyshev:2000] and [Fernando:1997].

The implicit QR algorithm applies a sequence of similarity transformations to the tridiagonal matrix T until its off-diagonal elements become negligible and the diagonal elements have converged to the eigenvalues of T [Golub_VanLoan:1996]. It consists of a bulge-chasing procedure that implicitly includes shifts and use plane rotations (e.g., Givens rotations) which preserve the tridiagonal form of T.

High performance in the implicit QR algorithm implemented in STATPACK is obtained by:

- restructuring the QR iterations with a wave-front algorithm to accumulate the Givens rotations for computing the eigenvectors [Lang:1998] [VanZee_etal:2011];
- using a "BLAS3" blocked algorithm to update the eigenvectors by these Givens rotations when possible [Lang:1998];
- using of a novel perfect shift strategy in the QR iterations inspired by the works of [Godunov_etal:1993] [Malyshev:2000] and [Fernando:1997] which reduces significantly the number of QR iterations needed for convergence for many symmetric tridiagonal matrices;
- and, finally, OpenMP parallelization [Demmel_etal:1993].

Subset computations are not possible with the QR algorithm, but it is possible to compute only all the eigenvalues or both all the eigenvalues and associated eigenvectors.

The bisection-inverse iteration or bisection-deflation methods are the preferred methods if you are only interested in a subset of the eigenvalues and eigenvectors of a (symmetric) tridiagonal matrix `T` or a full symmetric matrix `MAT`.

If the distance between the eigenvalues of `T` is sufficient relative to the (spectral of Frobenius) norm of `T`, then computing eigenvectors by inverse iteration is a $O(n.k)$ process, where `k` is the number of eigenvectors to compute [Ipsen:1997] [Dhillon:1998]. However, if the eigenvalues of `T` are too close, the eigenvectors must be orthogonalized by the modified Gram-Schmidt or QR algorithms, which are more expensive. However, when large clusters of eigenvalues are present, the use of a "BLAS3" and parallelized QR algorithm in the orthogonalization step increases significantly the efficiency of the algorithm. It is also recommended to compute the eigenvalues of `T` to high accuracy (e.g., by the bisection method implemented in *`symtrid_bisect()`* or *`select_eigval_cmp3()`* subroutines) for the success of the inverse iteration technique.

The deflation method combines Fernando's method for the computation of eigenvectors [Fernando:1997] [Parlett_Dhillon:1997] with deflation procedures by Givens rotations, see [Godunov_etal:1993] [Parlett_Dhillon:1997] [Malyshev:2000] for more details. QR iterations with a perfect shift strategy are also used as a back-up procedure if the deflation technique fails [Mastronardi_etal:2006]. If the eigenvalues are well-separated, the deflation method is also a $O(n.k)$ process, where `k` is the number of eigenvectors to compute. It is also highly recommended to compute the eigenvalues of `T` to high accuracy (e.g., again by the bisection method implemented in *`symtrid_bisect()`* or *`select_eigval_cmp3()`* subroutines) for the success of the deflation technique.

Parallelism concerns only the computation of eigenvectors in the QR method, but both the computation of the eigenvalues and eigenvectors in the bisection-inverse iteration and bisection-deflation methods.

Finally, as already explained above, subset computations are not possible in the standard implicit QR algorithm, but is possible with the two other methods for computing eigenvectors and for the bisection method for computing eigenvalues. The `m` largest or smallest eigenvalues of a symmetric `n`-by-`n` tridiagonal matrix `T` can also be computed using the rational QR method [Reinsch_Bauer:1968]. This rational QR method is however sequential in this version of STATPACK.

As a summary, in addition to a large set of computational routines for specific tasks related to the EVD of dense symmetric matrices, this module provides five EVD drivers for computing eigenvalues and associated eigenvectors of a `n`-by-`n` dense and real symmetric matrix `MAT`:

- *`eig_cmp()`*, *`eig_cmp2()`* and *`eig_cmp3()`* subroutines, which compute all the eigenvalues and eigenvectors of `MAT` using different fast variants of the implicit QR algorithm;
- *`eig_cmp4()`* and *`eig_cmp5()`* subroutines, which compute all or selected eigenvalues and associated eigenvectors of `MAT` using the fast Pal-Walker-Kahan variant of the implicit QR method or bisection for the eigenvalues and a fast inverse iteration strategy for computing the associated eigenvectors.

In most cases, *`eig_cmp4()`* and *`eig_cmp5()`* are significantly faster than *`eig_cmp()`*, *`eig_cmp2()`* and *`eig_cmp3()`*, but may be slightly less accurate and robust in a few cases.

Note, finally, that the driver and computational EVD routines based on the QR or bisection-inverse iterations provided in this module are different from the corresponding routines provided by LAPACK [Anderson_etal:1999] and are (much) faster if OpenMP and BLAS support are used, but sometimes slightly less accurate for the same precision.

In addition to these *standard* and *deterministic* drivers and computational routines based on implicit QR tridiagonal, inverse or deflation iterations applied to tridiagonal matrices after a preliminary tridiagonal reduction step, module *Eig_Procedures* also includes an optimized routine for computing an approximation of the largest eigenvalues (in absolute magnitude) and associated eigenvectors of full symmetric matrices using randomized power, subspace or block Krylov iterations [Halko_etal:2011] [Musco_Musco:2015] [Martinsson:2019]. Note also that module *SVD_Procedures* contains a similar optimized routine, *`reig_pos_cmp()`*, for real symmetric positive (semi-)definite matrices based on the so-called Nystrom method [Halko_etal:2011] [Martinsson:2019]. The Nystrom method provides more accurate results for positive semi-definite matrices.

For a good introduction to randomized linear algebra, see [Li_etal:2017], [Martinsson:2019] and [Erichson_etal:2019]. These randomized methods identify a subspace that captures most of the action (i.e. capture the largest singular values) of the symmetric matrix. The basic idea of these randomized methods is to use random projection to approximate the dominant subspace of a (symmetric) matrix. The input (symmetric) matrix is then compressed-either explicitly or implicitly to this subspace, and the *reduced* symmetric matrix is manipulated inexpensively by *standard* methods to obtain the desired low-rank factorization. In many cases, this approach beats largely its classical competitors in terms of speed [Halko_etal:2011] [Musco_Musco:2015] [Li_etal:2017]. Thus, these routines based on recent randomization algorithms are much faster than the *standard* and *deterministic* drivers included in module *Eig_Procedures* for computing a truncated EVD of a symmetric matrix. Yet, such randomized methods are also shown to compute with a very high probability low-rank approximations that are accurate, and are known to perform even better in many practical situations when the eigenvalues of the input symmetric matrix decay quickly [Halko_etal:2011] [Li_etal:2017].

The randomized algorithms included in module *Eig_Procedures* are also parallelized with OpenMP [openmp].

Please note that driver and computational routines provided in this module apply only to real data of kind **stnd**. The real kind type **stnd** is defined in module *Select_Parameters*. Computation of eigenvalues and eigenvectors for a complex matrix are not provided in this release of STATPACK.

In order to use one of these routines, you must include an appropriate `use Eig_Procedures` or `use Statpack` statement in your Fortran program, like:

```
use EIG_Procedures, only: eig_cmp
```

or :

```
use Statpack, only: eig_cmp
```

Here is the list and documentation of the public routines exported by module *EIG_Procedures*:

symtrid_cmp()

Purpose:

symtrid_cmp() reduces a real `n`-by-`n` symmetric matrix `MAT` (eventually stored in packed format) to symmetric tridiagonal form `T` by an orthogonal similarity transformation:

$$Q^T * MAT * Q = T$$

The transformation to tridiagonal form `T`, which uses Householder reflectors, is blocked and parallelized with OpenMP if the *UPPER* argument is not used [Dongarra_etal:1989].

On the other hand, if the *UPPER* argument is used, the standard sequential and non-blocked algorithm is used [Golub_VanLoan:1996] [Parlett:1998].

The matrix `Q` is stored as a product of elementary Householder reflectors in the lower or upper triangle of `MAT` on exit. Subroutines *ortho_gen_symtrid()* and *apply_q_symtrid()* can then be used to generate explicitly the orthogonal matrix `Q` or to apply it to an another matrix.

Synopsis:

```
call symtrid_cmp( mat(:n,:n) ,          d(:n) , e(:n) , store_q , upper )
call symtrid_cmp( mat(:n,:n) ,          d(:n) , e(:n) , store_q         )
call symtrid_cmp( matp(:(n*(n+1)/2)) , d(:n) , e(:n) , store_q , upper )
call symtrid_cmp( matp(:(n*(n+1)/2)) , d(:n) , e(:n) , store_q         )
```

Examples:

ex1_symtrid_cmp.F90

ex2_symtrid_cmp.F90

ex1_trid_inviter_bis.F90

ex2_trid_deflate.F90

symtrid_cmp2()

Purpose:

symtrid_cmp2() reduces a real n-by-n symmetric matrix cross-product

$$MAT^T * MAT$$

, where MAT is a m-by-n matrix, with m >= n, to symmetric tridiagonal form T by an orthogonal similarity transformation:

$$Q^T * MAT^T * MAT * Q = T$$

where Q is an orthogonal n-by-n matrix.

symtrid_cmp2() computes T and Q, using a parallel (if OpenMP support is activated) and blocked version of the one-sided Ralha tridiagonal reduction algorithm [Ralha:2003] [Hegland_etal:1999], without explicitly forming the matrix cross-product $MAT^T * MAT$.

The matrix Q is stored as a product of elementary Householder reflectors in the lower triangle of MAT on exit, if the input logical argument *STORE_Q* is set to the value true. Subroutines *ortho_gen_symtrid()* and *apply_q_symtrid()* (with logical argument *UPPER* set to false) can then be used to generate explicitly the orthogonal matrix Q or to apply it to an another matrix.

Synopsis:

```
call symtrid_cmp2( mat(:n,:n) , d(:n) , e(:n) , store_q )
```

Examples:

ex1_symtrid_cmp2.F90

ex2_symtrid_cmp2.F90

ortho_gen_symtrid()

Purpose:

ortho_gen_symtrid() generates a real orthogonal matrix Q, which is defined as the product of n-1 elementary reflectors of order n, as returned by *symtrid_cmp()* or *symtrid_cmp2()*.

The generation of the real orthogonal matrix Q in **ortho_gen_symtrid()** is blocked and parallelized with OpenMP in all cases using a method described in [Walker:1988].

Synopsis:

```
call ortho_gen_symtrid( mat(:n,:n) , upper )
call ortho_gen_symtrid( mat(:n,:n)         )
```

Examples:

ex1_symtrid_cmp.F90

ex2_symtrid_cmp.F90

ex1_symtrid_cmp2.F90

ex2_symtrid_cmp2.F90

apply_q_symtrid()

Purpose:

apply_q_symtrid() overwrites the general real m-by-n matrix C with:

- $Q * C$ if *LEFT* = true and *TRANS* = false ;

- $Q^T * C$ if *LEFT* = `true` and *TRANS* = `true` ;
- $C * Q$ if *LEFT* = `false` and *TRANS* = `false` ;
- $C * Q^T$ if *LEFT* = `false` and *TRANS* = `true` .

where `Q` is a real orthogonal matrix of order `nq` and is defined as the product of `nq-1` elementary reflectors

- $Q = H(nq-1) * ... * H(2) * H(1)$, if *UPPER* = `true` or is not used
- $Q = H(1) * H(2) * ... * H(nq-1)$, if *UPPER* = `false`

as returned by *symtrid_cmp()* or *symtrid_cmp2()*.

`Q` is of order `m` (=`nq`) and is the product of `m-1` reflectors if *LEFT* = true. `Q` is of order `n` (=`nq`) and is the product of `n-1` reflectors if *LEFT* = false.

The procedure is blocked and parallelized with OpenMP in all cases using a method described in [Walker:1988].

Synopsis:

```
call apply_q_symtrid( mat(:nq,:nq) ,          c(:,:) , left , trans , upper )
call apply_q_symtrid( mat(:nq,:nq) ,          c(:,:) , left , trans         )
call apply_q_symtrid( matp(:(nq*(nq+1)/2)) , c(:,:) , left , trans , upper )
call apply_q_symtrid( matp(:(nq*(nq+1)/2)) , c(:,:) , left , trans ,       )
```

The shape of `C` must verify:

- `size( C, 1 ) = nq` if *LEFT* = `true`
- `size( C, 2 ) = nq` if *LEFT* = `false`

symtrid_qri()

Purpose:

symtrid_qri() computes all eigenvalues and eigenvectors of a symmetric `n`-by-`n` tridiagonal matrix using the implicit QR method [Golub_VanLoan:1996] [Parlett:1998] [Greenbaum_Dongarra:1989] .

The eigenvalues and eigenvectors of a full symmetric matrix can also be found if *symtrid_cmp()* and *ortho_gen_symtrid()* have been used to reduce this matrix to tridiagonal form.

The computation of the eigenvectors are parallelized with OpenMP using a technique described in [Demmel_etal:1993].

If eigenvectors are not requested, **symtrid_qri()** computes all eigenvalues of a symmetric tridiagonal matrix using the fast Pal-Walker-Kahan variant of the QR algorithm, which does not use square roots [Parlett:1998].

Synopsis:

```
call symtrid_qri( d(:n) , e(:n) , failure , mat(:n,:n) , init_mat=init_mat ,␣
↪sort=sort , maxiter=maxiter )
call symtrid_qri( d(:n) , e(:n) , failure , sort=sort , maxiter=maxiter )
```

Examples:

ex1_symtrid_qri.F90

ex2_symtrid_qri.F90

symtrid_qri2()

Purpose:

symtrid_qri2() computes all eigenvalues and eigenvectors of a symmetric `n`-by-`n` tridiagonal matrix with a perfect shift strategy for the eigenvectors [Godunov_etal:1993] [Malyshev:2000] .

If the perfect shift strategy failed for some eigenvectors, these eigenvectors are computed with the implicit QR method as a back-up method [Golub_VanLoan:1996] [Parlett:1998].

The eigenvalues and eigenvectors of a full symmetric matrix can also be found if *symtrid_cmp()* and *ortho_gen_symtrid()* have been used to reduce this matrix to tridiagonal form.

Furthermore, the computation of the eigenvectors is blocked with a wave-front "BLAS3" algorithm for applying Givens rotations [Lang:1998] [VanZee_etal:2011] and parellelized with OpenMP [Demmel_etal:1993].

If eigenvectors are not requested, **symtrid_qri2()** computes all eigenvalues of a symmetric tridiagonal matrix using the fast Pal-Walker-Kahan variant of the QR algorithm, which does not use square roots [Parlett:1998].

symtrid_qri2() is much faster than *symtrid_qri()* for computing eigenvectors of large matrices, but may be slightly less efficient for small matrices.

Synopsis:

```
call symtrid_qri2( d(:n) , e(:n) , failure , mat(:n,:n) , init_mat=init_mat ,␣
↪sort=sort , maxiter=maxiter, max_francis_steps=max_francis_steps )
call symtrid_qri2( d(:n) , e(:n) , failure , sort=sort , maxiter=maxiter )
```

Examples:

ex1_symtrid_qri2.F90

ex2_symtrid_qri2.F90

symtrid_qri3()

Purpose:

symtrid_qri3() computes all eigenvalues and eigenvectors of a symmetric n-by-n tridiagonal matrix with the implicit QR method [Golub_VanLoan:1996] [Parlett:1998].

The eigenvalues and eigenvectors of a full symmetric matrix can also be found if *symtrid_cmp()* and *ortho_gen_symtrid()* have been used to reduce this matrix to tridiagonal form.

The computation of the eigenvectors is blocked with a wave-front "BLAS3" algorithm for applying Givens rotations [Lang:1998] [VanZee_etal:2011] and parellelized with OpenMP [Demmel_etal:1993].

If eigenvectors are not requested, **symtrid_qri3()** computes all eigenvalues of a symmetric tridiagonal matrix using the implicit QR method [Golub_VanLoan:1996] [Parlett:1998].

symtrid_qri3() is (slightly) slower than *symtrid_qri2()* for computing eigenvectors of large matrices, but may be more robust in a few cases.

Synopsis:

```
call symtrid_qri3( d(:n) , e(:n) , failure , mat(:n,:n) , init_mat=init_mat ,␣
↪sort=sort , maxiter=maxiter, max_francis_steps=max_francis_steps )
call symtrid_qri3( d(:n) , e(:n) , failure , sort=sort , maxiter=maxiter )
```

Examples:

ex1_symtrid_qri3.F90

symtrid_ratqri()

Purpose:

symtrid_ratqri() computes the m largest or smallest eigenvalues of a real symmetric n-by-n tridiagonal matrix using a rational QR method [Reinsch_Bauer:1968].

The m largest or smallest eigenvalues of a full symmetric matrix can also be found if *symtrid_cmp()* and *ortho_gen_symtrid()* have been used to reduce this matrix to tridiagonal form.

This subroutine is not parallelized, but allows computation of a subset of the eigenvalues of a symmetric n-by-n tridiagonal matrix, a possibility, which is not offered by *symtrid_qri()*, *symtrid_qri2()* and *symtrid_qri3()*.

Synopsis:

```
call symtrid_ratqri( d(:n) , e(:n) , m , failure , small=small , tol=tol )
```

Examples:

ex1_symtrid_ratqri.F90

symtrid_ratqri2()

Purpose:

symtrid_ratqri2() computes the largest or smallest eigenvalues of a symmetric n-by-n tridiagonal matrix in algebraic value whose sum (e.g., sum of the absolute values of the eigenvalues) exceeds a given value. A rational QR method is used [Reinsch_Bauer:1968].

The largest or smallest eigenvalues of a full symmetric matrix whose sum exceeds a given threshold in algebraic value can also be found, if *symtrid_cmp()* and *ortho_gen_symtrid()* have been used to reduce this matrix to tridiagonal form.

This subroutine is not parallelized, but allows computation of a subset of the eigenvalues of a symmetric n-by-n tridiagonal matrix, a possibility, which is not offered by *symtrid_qri()*, *symtrid_qri2()* and *symtrid_qri3()*.

Synopsis:

```
call symtrid_ratqri2( d(:n) , e(:n) , val , failure , m , small=small ,␣
↪tol=tol )
```

Examples:

ex1_symtrid_ratqri2.F90

symtrid_bisect()

Purpose:

symtrid_bisect() computes all or some of the largest or smallest eigenvalues of a real n-by-n symmetric tridiagonal matrix T using a bisection method [Golub_VanLoan:1996].

The full set, largest or smallest eigenvalues of a full symmetric matrix can also be found if *symtrid_cmp()* has been used to reduce this matrix to tridiagonal form.

This subroutine is parallelized if OpenMP is used and allows the computation of eigenvalues to high (relative) accuracy if desired (e.g., with the optional argument *ABSTOL* set to `sqrt(lamch("S"))` or `safmin`, where `safmin` is a public numerical constant exported by the *Num_Constants* module).

Synopsis:

```
call symtrid_bisect( d(:n) , e(:n) , neig , w(:n) , failure , small=small␣
↪, sort=sort , vector=vector , abstol=abstol , le=le , theta=theta ,␣
↪scaling=scaling , init=init )
```

Examples:

ex1_symtrid_bisect.F90

ex2_symtrid_bisect.F90

ex1_trid_inviter_bis.F90

ex2_trid_deflate.F90

eig_cmp()

Purpose:

eig_cmp() computes all eigenvalues and eigenvectors of a n-by-n real symmetric matrix MAT.

The matrix MAT is first transformed to tridiagonal form T, then the eigenvalues and the eigenvectors are computed by the QR implicit algorithm [Golub_VanLoan:1996].

The transformation to tridiagonal form T is blocked and parallelized with OpenMP if the *UPPER* argument is not used [Dongarra_etal:1989]. Otherwise, a sequential and non-blocked algorithm is used for this transformation [Golub_VanLoan:1996] [Parlett:1998].

The computation of the eigenvectors by the QR implicit algorithm is parallelized with OpenMP in all cases using a technique described in [Demmel_etal:1993].

Synopsis:

```
call eig_cmp( mat(:n,:n) , eigval(:n) , failure , upper , sort=sort ,␣
↪maxiter=maxiter )
call eig_cmp( mat(:n,:n) , eigval(:n) , failure ,         sort=sort ,␣
↪maxiter=maxiter )
```

Examples:

ex1_eig_cmp.F90

ex2_eig_cmp.F90

ex1_random_eig.F90

ex1_random_eig_pos.F90

ex1_random_eig_with_blas.F90

ex1_random_eig_pos_with_blas.F90

eig_cmp2()

Purpose:

eig_cmp2() computes all eigenvalues and eigenvectors of a n-by-n real symmetric matrix MAT.

The matrix MAT is first transformed to tridiagonal form T, then the eigenvalues are computed by the Pal-Walker-Kahan variant of the QR algorithm [Parlett:1998] and the eigenvectors are computed with a perfect shift strategy [Godunov_etal:1993] [Malyshev:2000], or the implicit QR algorithm [Parlett:1998] if the perfect shift strategy fails.

The transformation to tridiagonal form T is blocked and parallelized with OpenMP if the *UPPER* argument is not used [Dongarra_etal:1989]. Otherwise, a sequential and non-blocked algorithm is used [Golub_VanLoan:1996] [Parlett:1998].

Furthermore, the computation of the eigenvectors is also blocked with a wave-front "BLAS3" algorithm for applying the Givens rotations which are used in the perfect shift strategy or the implicit QR algorithm (see [Lang:1998] [VanZee_etal:2011]) and is also parellelized with OpenMP in all cases [Demmel_etal:1993].

eig_cmp2() is much faster than *eig_cmp()* for large matrices, but may be slightly less efficient for small matrices.

Synopsis:

```
call eig_cmp2( mat(:n,:n) , eigval(:n) , failure , upper , sort=sort ,␣
↪maxiter=maxiter , max_francis_steps=max_francis_steps )
call eig_cmp2( mat(:n,:n) , eigval(:n) , failure ,         sort=sort ,␣
↪maxiter=maxiter , max_francis_steps=max_francis_steps )
```

Examples:

ex1_eig_cmp2.F90

ex2_eig_cmp2.F90

eig_cmp3()

Purpose:

eig_cmp3() computes all eigenvalues and eigenvectors of a n-by-n real symmetric matrix MAT.

The matrix MAT is first transformed to tridiagonal form T, then the eigenvalues/eigenvectors are computed by the implicit QR algorithm [Parlett:1998]. The Givens rotations used in the implicit QR algorithm are accumulated with a fast wavefront "BLAS3" algorithm for computing the eigenvectors [Lang:1998] [VanZee_etal:2011].

The transformation to tridiagonal form T is blocked and parallelized with OpenMP if the *UPPER* argument is not used [Dongarra_etal:1989]. Otherwise, a sequential and non-blocked algorithm is used [Golub_VanLoan:1996] [Parlett:1998].

Furthermore, the computation of the eigenvectors is also blocked with a wave-front "BLAS3" algorithm for applying the Givens rotations (see [Lang:1998] [VanZee_etal:2011]) and parellelized with OpenMP [Demmel_etal:1993].

eig_cmp3() is much faster than *eig_cmp()* for large matrices and less faster than *eig_cmp2()*, but may be more robust in a few cases.

Synopsis:

```
call eig_cmp3( mat(:n,:n) , eigval(:n) , failure , upper , sort=sort ,␣
↪maxiter=maxiter , max_francis_steps=max_francis_steps )
call eig_cmp3( mat(:n,:n) , eigval(:n) , failure ,         sort=sort ,␣
↪maxiter=maxiter , max_francis_steps=max_francis_steps )
```

Examples:

ex1_eig_cmp3.F90

ex2_eig_cmp3.F90

eig_cmp4()

Purpose:

eig_cmp4() computes all eigenvalues and eigenvectors of a n-by-n real symmetric matrix MAT.

The matrix MAT is first transformed to tridiagonal form T, then the eigenvalues of T are computed by the fast Pal-Walker-Kahan variant of the QR algorithm [Parlett:1998], the eigenvectors of T are computed with an inverse iteration strategy and are, finally, backed transformed into those of MAT [Golub_VanLoan:1996] [Parlett:1998].

The transformation to tridiagonal form T is blocked and parallelized with OpenMP [Dongarra_etal:1989].

Furthermore, the computation of the eigenvectors of T and their back transformation to those of MAT are also blocked and parallelized in all cases.

eig_cmp4() is usually much faster than the *eig_cmp()*, *eig_cmp2()* and *eig_cmp3()* drivers based on QR iterations for large matrices, but may be slightly less accurate in some cases.

Synopsis:

```
call eig_cmp4( mat(:n,:n) , eigval(:n) , failure ,  maxiter_qr=maxiter_qr ,
↪ maxiter_inviter=maxiter_inviter , ortho=ortho , backward_sweep=backward_
↪sweep , scaling=scaling , initvec=initvec )
```

Examples:

ex1_eig_cmp4.F90

eig_cmp5()

Purpose:

eig_cmp5() computes all or, some of the largest or smallest, eigenvalues and associated eigenvectors of a `n`-by-`n` real symmetric matrix `MAT`.

The matrix `MAT` is first transformed to tridiagonal form `T`, then the eigenvalues of `T` are computed by the bisection method [Golub_VanLoan:1996] [Parlett:1998], the eigenvectors of `T` are computed with an inverse iteration strategy and are, finally, backed transformed into those of `MAT` [Golub_VanLoan:1996] [Parlett:1998].

The transformation to tridiagonal form `T` is blocked and parallelized with OpenMP [Dongarra_etal:1989].

The full set, largest or smallest eigenvalues of the tridiagonal form `T` are computed by a parallel bisection algorithm if OpenMP is used. Furthermore, the subroutine allows the computation of eigenvalues of `T` to high (relative) accuracy if desired (e.g., with the optional argument *ABSTOL* set to `sqrt(lamch("S"))` or `safmin`, where `safmin` is a public numerical constant exported by the *Num_Constants* module).

Finally, the computation of the eigenvectors of `T` and their back transformation to those of `MAT` are also blocked and parallelized in all cases.

eig_cmp5() is usually much faster than the *eig_cmp()*, *eig_cmp2()* and *eig_cmp3()* drivers based on QR iterations for large matrices. **eig_cmp5()** is also more accurate than *eig_cmp4()* thanks to the use of the bisection algorithm for computing the eigenvalues, but may be sligthly slower for huge symmetric matrices. It is also more flexible than *eig_cmp4()* as subset computations are possible with **eig_cmp5()**, but not with *eig_cmp4()*.

Synopsis:

```
call eig_cmp5( mat(:n,:n) , neig , eigval(:n) , failure , small=small , le=le␣
↪, theta=theta , abstol=abstol , scaling_bisect=scaling_bisect , maxiter_
↪inviter=maxiter_inviter , ortho=ortho , backward_sweep=backward_sweep ,␣
↪scaling_inviter=scaling_inviter , initvec=initvec )
```

Examples:

ex1_eig_cmp5.F90

laev2()

Purpose:

laev2() computes the eigendecomposition of a `2`-by-`2` real symmetric matrix

$$\begin{pmatrix} a & b \\ b & c \end{pmatrix}$$

On return, `RT1` is the eigenvalue of larger absolute value, `RT2` is the eigenvalue of smaller absolute value, and `(CS1, SN1)` is the unit right eigenvector for `RT1`, giving the decomposition

$$\begin{pmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{pmatrix} \begin{pmatrix} a & b \\ b & c \end{pmatrix} \begin{pmatrix} cs1 & -sn1 \\ sn1 & cs1 \end{pmatrix} = \begin{pmatrix} rt1 & O \\ 0 & rt2 \end{pmatrix}$$

This subroutine is adapted from LAPACK [Anderson_etal:1999].

Synopsis:

```
call laev2( a , b , c , rt1 , rt2 , cs1 , sn1 )
```

reig_cmp()

Purpose:

reig_cmp() computes approximations of the `neig` largest eigenvalues (in absolute magnitude) and associated eigenvectors of a full real `n`-by-`n` symmetric matrix `MAT` using randomized power, subspace or block Krylov iterations [Halko_etal:2011] [Musco_Musco:2015] [Martinsson:2019].

This routine is always significantly faster than *eig_cmp()*, *eig_cmp2()*, *eig_cmp3()*, *trid_inviter()* or *trid_deflate()* because of the use of very fast randomized algorithms [Halko_etal:2011] [Gu:2015] [Musco_Musco:2015] [Martinsson:2019]. However, the computed `neig` largest eigenvalues (in absolute magnitude) and associated eigenvectors are only approximations of the *true* largest eigenvalues and eigenvectors.

Note that module *SVD_Procedures* contains a similar routine, *reig_pos_cmp()*, for real symmetric positive semi-definite matrices based on the so-called Nystrom method [Halko_etal:2011] [Martinsson:2019]. The Nystrom method provides more accurate results for positive semi-definite matrices, so if your input matrix `MAT` is positive semi-definite, *reig_pos_cmp()* is a better choice than **reig_cmp()**.

The routine returns approximations to the first `neig` eigenvalues (in absolute magnitude) and the associated eigenvectors corresponding to a partial EVD of a symmetric matrix `MAT`.

Synopsis:

```
call reig_cmp( mat(:n,:n) , eigval(:neig) , eigvec(:n,:neig) ,␣
↪failure=failure , niter=niter , nover=nover , ortho=ortho , extd_samp=extd_
↪samp , rng_alg=rng_alg , maxiter=maxiter )
```

Examples:

ex1_reig_cmp.F90

eig_sort()

Purpose:

Given the eigenvalues *D* and, eventually, eigenvectors *U* as output from *eig_cmp()*, *eig_cmp2()*, *eig_cmp3()*, *symtrid_qri()*, *symtrid_qri2()*, *symtrid_qri3()* or other STATPACK routines which compute eigenvalues and eigenvectors, **eig_sort()** sorts the eigenvalues *D* into ascending or descending order, and, rearranges the columns of *U* correspondingly.

Synopsis:

```
call eig_sort( sort , d(:m), u(:n,:m) )
call eig_sort( sort , d(:m)           )
```

eig_abs_sort()

Purpose:

Given the eigenvalues *D* and, eventually, eigenvectors *U* as output from *eig_cmp()*, *eig_cmp2()*, *eig_cmp3()*, *symtrid_qri()*, *symtrid_qri2()*, *symtrid_qri3()* or other STATPACK routines which compute eigenvalues and eigenvectors, **eig_abs_sort()** sorts the eigenvalues *D* into ascending or descending order of absolute magnitude, and, rearranges the columns of *U* correspondingly.

Synopsis:

```
call eig_abs_sort( sort , d(:m), u(:n,:m) )
call eig_abs_sort( sort , d(:m)           )
```

Examples:

ex1_random_eig.F90

ex1_random_eig_pos.F90

ex1_random_eig_pos_with_blas.F90

eigvalues()

Purpose:

eigvalues() computes all eigenvalues of a n-by-n real symmetric matrix MAT.

The matrix MAT is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

$$Q^T * MAT * Q = T$$

then the eigenvalues of T, which are also the eigenvalues of MAT, are computed by the Pal-Walker-Kahan variant of the QR algorithm [Parlett:1998].

If the QR algorithm fails to converge **eigvalues()** returns a n-vector filled with NANs.

Synopsis:

```
eigval(:n) = eigvalues( mat(:n,:n) , upper , sort=sort , maxiter=maxiter  )
eigval(:n) = eigvalues( mat(:n,:n) ,         sort=sort , maxiter=maxiter  )
```

Examples:

ex1_eigvalues.F90

eigval_cmp()

Purpose:

eigval_cmp() computes all eigenvalues of a n-by-n real symmetric matrix MAT, eventually stored in packed form.

The matrix MAT is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

$$Q^T * MAT * Q = T$$

then the eigenvalues of T, which are also the eigenvalues of MAT, are computed by the Pal-Walker-Kahan variant of the QR algorithm [Parlett:1998].

Synopsis:

```
call eigval_cmp( mat(:n,:n) ,         eigval(:n) , failure , upper ,␣
↪sort=sort , maxiter=maxiter , d_e=d_e(:n,:2) )
call eigval_cmp( mat(:n,:n) ,         eigval(:n) , failure ,        ␣
↪sort=sort , maxiter=maxiter , d_e=d_e(:n,:2) )
call eigval_cmp( matp(:(n*(n+1)/2)) , eigval(:n) , failure , upper ,␣
↪sort=sort , maxiter=maxiter , d_e=d_e(:n,:2) )
call eigval_cmp( matp(:(n*(n+1)/2)) , eigval(:n) , failure ,        ␣
↪sort=sort , maxiter=maxiter , d_e=d_e(:n,:2) )
```

Examples:

ex1_eigval_cmp.F90

ex2_eigval_cmp.F90

eigval_cmp2()

Purpose:

eigval_cmp2() computes all eigenvalues of a n-by-n real symmetric matrix MAT, eventually stored in packed form.

The matrix MAT is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

$$Q^T * MAT * Q = T$$

then the eigenvalues of T, which are also the eigenvalues of MAT, are computed by the Pal-Walker-Kahan variant of the QR algorithm [Parlett:1998].

A slightly different version of the Pal-Walker-Kahan algorithm is used compared to the *eigval_cmp()* generic subroutine.

Synopsis:

```
call eigval_cmp2( mat(:n,:n) ,          eigval(:n) , failure , upper ,␣
↪sort=sort , maxiter=maxiter , d_e=d_e(:n,:2) )
call eigval_cmp2( mat(:n,:n) ,          eigval(:n) , failure ,        ␣
↪sort=sort , maxiter=maxiter , d_e=d_e(:n,:2) )
call eigval_cmp2( matp(:(n*(n+1)/2)) , eigval(:n) , failure , upper ,␣
↪sort=sort , maxiter=maxiter , d_e=d_e(:n,:2) )
call eigval_cmp2( matp(:(n*(n+1)/2)) , eigval(:n) , failure ,        ␣
↪sort=sort , maxiter=maxiter , d_e=d_e(:n,:2) )
```

Examples:

ex1_eigval_cmp2.F90

ex2_eigval_cmp2.F90

eigval_cmp3()

Purpose:

eigval_cmp3() computes all eigenvalues of a n-by-n real symmetric matrix MAT, eventually stored in packed form.

The matrix MAT is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

$$Q^T * MAT * Q = T$$

then the eigenvalues of T, which are also the eigenvalues of MAT, are computed by the implicit QR algorithm [Parlett:1998] [Golub_VanLoan:1996] .

Synopsis:

```
call eigval_cmp3( mat(:n,:n) ,          eigval(:n) , failure , upper ,␣
↪sort=sort , maxiter=maxiter , d_e=d_e(:n,:2) )
call eigval_cmp3( mat(:n,:n) ,          eigval(:n) , failure ,        ␣
↪sort=sort , maxiter=maxiter , d_e=d_e(:n,:2) )
call eigval_cmp3( matp(:(n*(n+1)/2)) , eigval(:n) , failure , upper ,␣
↪sort=sort , maxiter=maxiter , d_e=d_e(:n,:2) )
call eigval_cmp3( matp(:(n*(n+1)/2)) , eigval(:n) , failure ,        ␣
↪sort=sort , maxiter=maxiter , d_e=d_e(:n,:2) )
```

Examples:

ex1_eigval_cmp3.F90

ex2_eigval_cmp3.F90

select_eigval_cmp()

Purpose:

select_eigval_cmp() computes the m = size(eigval) largest or smallest eigenvalues of a n-by-n real symmetric matrix MAT. MAT can be stored in packed form.

The matrix MAT is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

$$Q^T * MAT * Q = T$$

then the eigenvalues of T, which are also the eigenvalues of MAT, are computed by a rational QR method [Reinsch_Bauer:1968].

Synopsis:

```
call select_eigval_cmp( mat(:n,:n) ,          eigval(:m) , small , failure ,␣
↪upper , d_e=d_e(:n,:2) )
call select_eigval_cmp( mat(:n,:n) ,          eigval(:m) , small , failure ,  ␣
↪      d_e=d_e(:n,:2) )
call select_eigval_cmp( matp(:(n*(n+1)/2)) , eigval(:m) , small , failure ,␣
↪upper , d_e=d_e(:n,:2) )
call select_eigval_cmp( matp(:(n*(n+1)/2)) , eigval(:m) , small , failure ,  ␣
↪      d_e=d_e(:n,:2) )
```

Examples:

ex1_select_eigval_cmp.F90

ex2_select_eigval_cmp.F90

select_eigval_cmp2()

Purpose:

select_eigval_cmp2() computes the largest or smallest eigenvalues of a n-by-n real symmetric matrix MAT whose sum in algebraic value (e.g., the sum of the absolute values) exceeds a given value. MAT can be stored in packed form.

The matrix MAT is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

$$Q^T * MAT * Q = T$$

then the eigenvalues of T, which are also the eigenvalues of MAT, are computed by a rational QR method [Reinsch_Bauer:1968].

Synopsis:

```
call select_eigval_cmp2( mat(:n,:n) ,          eigval(:m) , small , val ,␣
↪failure , upper , d_e=d_e(:n,:2) )
call select_eigval_cmp2( mat(:n,:n) ,          eigval(:m) , small , val ,␣
↪failure ,         d_e=d_e(:n,:2) )
call select_eigval_cmp2( matp(:(n*(n+1)/2)) , eigval(:m) , small , val ,␣
↪failure , upper , d_e=d_e(:n,:2) )
call select_eigval_cmp2( matp(:(n*(n+1)/2)) , eigval(:m) , small , val ,␣
↪failure ,         d_e=d_e(:n,:2) )
```

Examples:

ex1_select_eigval_cmp2.F90

ex2_select_eigval_cmp2.F90

select_eigval_cmp3()

Purpose:

select_eigval_cmp3() computes the largest or smallest eigenvalues of a n-by-n real symmetric matrix MAT, eventually stored in packed form.

The matrix MAT is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

$$Q^T * MAT * Q = T$$

then the eigenvalues of T, which are also the eigenvalues of MAT, are computed by a bisection method [Golub_VanLoan:1996].

The eigenvalues of T can be computed to high accuracy with the optional argument *ABSTOL* set to sqrt(lamch("S")) or safmin, where safmin is a public numerical constant exported by the *Num_Constants* module.

Synopsis:

```
call select_eigval_cmp3( mat(:n,:n) ,          neig , eigval(:n) , small ,␣
↪failure , upper , sort=sort , vector=vector , scaling=scaling , init=init ,␣
↪abstol=abstol , le=le , theta=theta , d_e=d_e(:n,:2) )
call select_eigval_cmp3( mat(:n,:n) ,          neig , eigval(:n) , small ,␣
↪failure ,         sort=sort , vector=vector , scaling=scaling , init=init ,␣
↪abstol=abstol , le=le , theta=theta , d_e=d_e(:n,:2) )
call select_eigval_cmp3( matp(:(n*(n+1)/2)) , neig , eigval(:n) , small ,␣
↪failure , upper , sort=sort , vector=vector , scaling=scaling , init=init ,␣
↪abstol=abstol , le=le , theta=theta , d_e=d_e(:n,:2) )
call select_eigval_cmp3( matp(:(n*(n+1)/2)) , neig , eigval(:n) , small ,␣
↪failure ,         sort=sort , vector=vector , scaling=scaling , init=init ,␣
↪abstol=abstol , le=le , theta=theta , d_e=d_e(:n,:2) )
```

Examples:

ex1_select_eigval_cmp3.F90

ex2_select_eigval_cmp3.F90

lae2()

Purpose:

lae2() computes the eigenvalues of a 2-by-2 symmetric matrix

$$\begin{pmatrix} a & b \\ b & c \end{pmatrix}$$

On return, RT1 is the eigenvalue of larger absolute value, RT2 is the eigenvalue of smaller absolute value.

This subroutine is adapted from LAPACK [Anderson_etal:1999].

Synopsis:

```
call lae2( a , b , c , rt1 , rt2 )
```

eigval_sort()

Given the eigenvalues *D* as output from *eigval_cmp()*, *eigval_cmp2()*, *eigval_cmp3()*, *symtrid_qri()*, *symtrid_qri2()*, *symtrid_qri3()* or other STATPACK routines which compute eigenvalues, **eigval_sort()** sorts the eigenvalues *D* into ascending or descending order.

Synopsis:

```
call eigval_sort( sort , d(:m) )
```

Examples:

ex1_reig_pos_cmp.F90

ex1_random_eig_pos.F90

eigval_abs_sort()

Given the eigenvalues *D* as output from *eigval_cmp()*, *eigval_cmp2()*, *eigval_cmp3()*, *symtrid_qri()*, *symtrid_qri2()*, *symtrid_qri3()* or other STATPACK routines which compute eigenvalues, **eigval_abs_sort()** sorts the eigenvalues *D* into ascending or descending order of absolute magnitude.

Synopsis:

```
call eigval_abs_sort( sort , d(:m) )
```

dflgen()

Purpose:

dflgen() computes deflation parameters (e.g., a chain of Givens rotations) for a n-by-n symmetric unreduced tridiagonal matrix T and a given eigenvalue *LAMBDA* of T.

On output, the arguments *CS* and *SN* contain, respectively, the vectors of the cosines and sines coefficients of the chain of n-1 planar rotations that deflates the real n-by-n symmetric tridiagonal matrix T corresponding to the eigenvalue *LAMBDA*.

See [Malyshev:2000] for more details.

Synopsis:

```
call dflgen( d(:n) , e(:n-1) , lambda , cs(:n-1) , sn(:n-1) )
```

dflgen2()

Purpose:

dflgen2() computes and applies deflation parameters (e.g., a chain of Givens rotations) for a n-by-n symmetric unreduced tridiagonal matrix T and a given eigenvalue *LAMBDA* of T.

On output:

- the arguments *D* and *E* contain, respectively, the new main diagonal and off-diagonal of the deflated symmetric tridiagonal matrix if *DEFLATE* is set to true.
- the arguments *CS* and *SN* contain, respectively, the vectors of the cosines and sines coefficients of the chain of n-1 planar rotations that deflates the real n-by-n symmetric tridiagonal matrix T corresponding to the eigenvalue *LAMBDA*.

See [Malyshev:2000] for more details.

Synopsis:

```
call dflgen2( d(:n) , e(:n-1) , lambda , cs(:n-1) , sn(:n-1) , deflate )
```

dflapp()

Purpose:

dflapp() deflates a n-by-n real symmetric tridiagonal matrix T by a chain of planar rotations produced by *dflgen()*.

On output, the arguments *D* and *E* contain, respectively, the new main diagonal and off-diagonal of the deflated symmetric tridiagonal matrix, if *DEFLATE* is set to true on output.

See [Malyshev:2000] for more details.

Synopsis:

```
call dflapp( d(:n) , e(:n-1) , cs(:n-1) , sn(:n-1) , deflate )
```

qrstep()

Purpose:

qrstep() performs one QR step with a given shift *LAMBDA* on a n-by-n real symmetric unreduced tridiagonal matrix T.

On output, the arguments *D* and *E* contain, respectively, the new main diagonal and off-diagonal of the deflated symmetric tridiagonal. The chain of n-1 planar rotations produced during the QR step are saved in the arguments *CS* and *SN*. See [Mastronardi_etal:2006] for more details.

Synopsis:

```
call qrstep( d(:n) , e(:n-1) , lambda , cs(:n-1) , sn(:n-1) , deflate )
```

prodgiv()

Purpose:

prodgiv() applies a chain of `n-1` planar rotations produced by *dflgen()*, *dflgen2()* or *qrstep()* to the vector argument *X* of length `n`. See [Mastronardi_etal:2006] for more details.

Synopsis:

```
call prodgiv( cs(:n-1) , sn(:n-1) , x(:n) )
```

prodgiv_eigvec()

Purpose:

prodgiv_eigvec() computes an *approximate* eigenvector of a `n`-by-`n` symmetric tridiagonal matrix from a chain of `n-1` planar rotations produced by *dflgen()*, *dflgen2()* or *qrstep()*. See [Mastronardi_etal:2006] for more details.

Synopsis:

```
eigvec(:n) = prodgiv_eigvec( cs(:n-1) , sn(:n-1) )
```

symtrid_deflate()

Purpose:

symtrid_deflate() computes eigenvectors of a real symmetric tridiagonal matrix `T` corresponding to specified eigenvalues, using sequential deflation techniques [Godunov_etal:1993] [Malyshev:2000] [Mastronardi_etal:2006].

symtrid_deflate() may fail if some the eigenvalues specified in parameter *EIGVAL* are nearly identical or for clusters of small eigenvalues or for some zero-diagonal tridiagonal matrices.

symtrid_deflate() is a low-level routine used by the *trid_deflate()* driver routine. Its direct use as a method for computing eigenvectors of a real symmetric tridiagonal matrix is not recommended.

Synopsis:

```
call symtrid_deflate( d(:n) , e(:n-1) , eigval     , eigvec(:n)    , failure ␣
↪    , max_qr_steps=max_qr_steps )
call symtrid_deflate( d(:n) , e(:n-1) , eigval(:p) , eigvec(:n,:p) ,␣
↪failure(:p) , max_qr_steps=max_qr_steps )
```

trid_deflate()

Purpose:

trid_deflate() computes all or selected eigenvectors of a `n`-by-`n` real symmetric tridiagonal `T` or full symmetric matrix `MAT` (eventually packed column-wise in a linear array *MATP*) corresponding to specified eigenvalues, using deflation techniques [Godunov_etal:1993] [Malyshev:2000] [Mastronardi_etal:2006].

If eigenvectors of a full symmetric matrix `MAT` are wanted, it is required that the original symmetric matrix `MAT` has been reduced to symmetric tridiagonal form `T` by an orthogonal similarity transformation:

$$Q^T * MAT * Q = T$$

with a call to *symtrid_cmp()* with parameter *STORE_Q* set to `true`, before calling **trid_deflate()**.

The eigenvectors of `T` are computed using an efficient approach for the computation of (selected) eigenvectors of a tridiagonal matrix corresponding to (selected) eigenvalues by combining Fernando's method for the computation of eigenvectors with deflation procedures by Givens rotations [Fernando:1997] [Parlett_Dhillon:1997] [Malyshev:2000]. QR iterations are also used as a back-up procedure if the deflation technique fails [Mastronardi_etal:2006].

If eigenvectors of a full symmetric matrix `MAT` are wanted, the computed eigenvectors of `T` are backed-transformed to the eigenvectors of `MAT` using the packed representation of `Q` as computed by *symtrid_cmp()*.

Both the computation of the eigenvectors of `T` and the back-transformation phase are parallelized with OpenMP [Walker:1988].

It is essential that eigenvalues given on entry of **trid_deflate()** are computed to high (relative) accuracy. Subroutine *symtrid_bisect()* or *select_eigval_cmp3()* may be used for this purpose.

trid_deflate() may fail if some the eigenvalues specified in parameter *EIGVAL* are nearly identical for some pathological matrices or for clusters of small eigenvalues or for some zero-diagonal matrices.

The deflation algorithms used in **trid_deflate()** are competitive with the inverse iteration procedure implemented in *trid_inviter()* subroutine when the eigenvalues are well separated, otherwise *trid_inviter()* subroutine will be faster because of the use of a fast "BLAS3" QR algorithm for the reorthogonalization of the eigenvectors in *trid_inviter()*.

Synopsis:

```
call trid_deflate( d(:n) , e(:n) , eigval(:p) , eigvec(:n,:p) , failure ,␣
↪                  max_qr_steps=max_qr_steps , ortho=ortho , inviter=inviter )
call trid_deflate( d(:n) , e(:n) , eigval(:p) , eigvec(:n,:p) , failure␣
↪, mat(:n,:n)          , max_qr_steps=max_qr_steps , ortho=ortho ,␣
↪inviter=inviter )
call trid_deflate( d(:n) , e(:n) , eigval(:p) , eigvec(:n,:p) , failure␣
↪, matp(:(n*(n+1)/2)) , max_qr_steps=max_qr_steps , ortho=ortho ,␣
↪inviter=inviter )
```

Examples:

ex1_trid_deflate.F90

ex2_trid_deflate.F90

ex3_trid_deflate.F90

maxdiag_tinv_qr()

Purpose:

maxdiag_tinv_qr() computes the index of the element of maximum absolute value in the diagonal entries of the matrix

$$(T - lambda * I)^{-1}$$

where `T` is a `n`-by-`n` symmetric tridiagonal matrix, `I` is the identity matrix and `lambda` is a scalar.

The diagonal entries of `( T - lambda * I )`$^{-1}$ are computed by means of the QR factorization of `T - lambda * I`.

For more details, see [Bini_etal:2005].

It is assumed that `T` is unreduced, but no check is done in the subroutine to verify this assumption.

Synopsis:

```
maxdiag_tinv = maxdiag_tinv_qr( d(:n) , e(:n-1) , lambda )
```

maxdiag_tinv_ldu()

Purpose:

maxdiag_tinv_ldu() computes the index of the element of maximum absolute value in the diagonal entries of

$$(T - lambda * I)^{-1}$$

where `T` is a `n`-by-`n` symmetric tridiagonal matrix, `I` is the identity matrix and `lambda` is a scalar.

The diagonal entries of `( T - lambda * I )`$^{-1}$ are computed by means of LDU and UDL factorizations of `T - lambda * I`.

For more details, see [Fernando:1997].

It is assumed that T is unreduced, but no check is done in the subroutine to verify this assumption.

Synopsis:

```
maxdiag_tinv = maxdiag_tinv_ldu( d(:n) , e(:n-1) , lambda )
```

trid_qr_cmp()

Purpose:

trid_qr_cmp() factorizes the symmetric matrix T - lambda * I, where T is a n-by-n symmetric tridiagonal matrix, I is the identity matrix and lambda is a scalar, as

$$T - lambda * I = Q * R$$

where Q is an orthogonal matrix represented as the product of n-1 Givens rotations and R is an upper triangular matrix with at most two non-zero super-diagonal elements per column.

The parameter lambda is included in the routine so that **trid_qr_cmp()** may be used to obtain eigenvectors of T by inverse iteration.

The subroutine also computes the index of the entry of maximum absolute value in the diagonal of (T - lambda * I)$^{-1}$, which provides a good initial approximation to start the inverse iteration process for computing the eigenvector associated with the eigenvalue lambda.

For further details, see [Bini_etal:2005] [Fernando:1997] [Parlett_Dhillon:1997].

Synopsis:

```
call trid_qr_cmp( d(:n) , e(:n-1) , lambda , cs(:n-1) , sn(:n-1) , diag(:n) ,␣
↪sup1(:n) , sup2(:n) , maxdiag_tinv )
```

trid_qr_solve()

Purpose:

trid_qr_solve() may be used to solve for the vector x in the system of equations

$$x(:) * (T - lambda * I) = scale * y(:)$$

where T is a n-by-n symmetric tridiagonal matrix, I is the n-by-n identity matrix, lambda and scale are scalars, following the factorization of (T - lambda * I) by *trid_qr_cmp()* or *gk_qr_cmp()*, as

$$T - lambda * I = Q * R$$

where Q is an orthogonal matrix represented as the product of n-1 Givens rotations and R is an upper triangular matrix with at most two non-zero super-diagonal elements per column.

The matrix (T - lambda * I) is assumed to be ill-conditioned, and frequent rescalings are carried out in order to avoid overflow. However, no test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine. The scalar scale is not output by this routine since this routine being intended for use in applications such as inverse iteration.

Synopsis:

```
call trid_qr_solve( cs(:n-1) , sn(:n-1) , diag(:n) , sup1(:n) , sup2(:n) ,␣
↪y(:n) )
```

trid_cmp()

Purpose:

trid_cmp() factorizes the symmetric matrix $(T - eigval * I)$, where T is a n-by-n symmetric tridiagonal matrix, I is the n-by-n identity matrix, eigval is a scalar, as

$$T - eigval * I = P * L * U$$

where P is a permutation matrix, L is a unit lower tridiagonal matrix with at most one non-zero sub-diagonal elements per column and U is an upper triangular matrix with at most two non-zero super-diagonal elements per column.

The factorizations is obtained by Gaussian elimination with partial pivoting and implicit row scaling.

Several symmetric matrices $(T - eigval_i * I)$ can be handled in a single call.

The parameter *EIGVAL* is included in the routine so that **trid_cmp()** may be used to obtain eigenvectors of T by inverse iteration.

Synopsis:

```
call trid_cmp( d(:n) , e(:n) , eigval     , sub(:n)    , diag(:n)    ,␣
↪sup1(:n)    , sup2(:n)    , perm(:n)    , tol=tol )
call trid_cmp( d(:n) , e(:n) , eigval(:p) , sub(:p,:n) , diag(:p,:n) ,␣
↪sup1(:p,:n) , sup2(:p,:n) , perm(:p,:n) , tol=tol )
```

trid_cmp2()

Purpose:

trid_cmp2() factorizes the symmetric matrix $(T - eigval * I)$, where T is a n-by-n symmetric tridiagonal matrix, I is the n-by-n identity matrix, eigval is a scalar, as

$$T - eigval * I = P * L * U$$

where P is a permutation matrix, L is a unit lower tridiagonal matrix with at most one non-zero sub-diagonal elements per column and U is an upper triangular matrix with at most two non-zero super-diagonal elements per column.

The factorization is obtained by Gaussian elimination with partial pivoting and row interchanges.

Several symmetric matrices $(T - eigval_i * I)$ can be handled in a single call.

The parameter *EIGVAL* is included in the routine so that **trid_cmp2()** may be used to obtain eigenvectors of T by inverse iteration.

trid_cmp2() is a simplified version of *trid_cmp()*.

Synopsis:

```
call trid_cmp2( d(:n) , e(:n) , eigval     , sub(:n)    , diag(:n)    ,␣
↪sup1(:n)    , sup2(:n)    , perm(:n)    )
call trid_cmp2( d(:n) , e(:n) , eigval(:p) , sub(:p,:n) , diag(:p,:n) ,␣
↪sup1(:p,:n) , sup2(:p,:n) , perm(:p,:n) )
```

trid_solve()

Purpose:

trid_solve() may be used to solve for a vector x in the system of equations

$$x * (T - eigval * I) = scale * y$$

where T is a n-by-n symmetric tridiagonal matrix, I is the n-by-n identity matrix, eigval and scale are scalars and y is a vector, following the factorization of $(T - eigval * I)$ by *trid_cmp()* or *trid_cmp2()* as

$$T - eigval * I = P * L * U$$

where P is a permutation matrix, L is a unit lower tridiagonal matrix with at most one non-zero sub-diagonal elements per column and U is an upper triangular matrix with at most two non-zero super-diagonal elements per column.

The matrix $(T - eigval * I)$ is assumed to be ill-conditioned, and frequent rescalings are carried out in order to avoid overflow. However, no test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

Several systems $(T - eigval_i * I)$ can be handled in a single call.

The scalar `scale` is not output by this routine since this routine being intended for use in applications such as inverse iteration.

Synopsis:

```
call trid_solve( sub(:n)    , diag(:n)    , sup1(:n)    , sup2(:n)    ,␣
↪perm(:n)    , y(:n)    )
call trid_solve( sub(:p,:n) , diag(:p,:n) , sup1(:p,:n) , sup2(:p,:n) ,␣
↪perm(:p,:n) , y(:p,:n) )
```

trid_inviter()

Purpose:

trid_inviter() computes all or selected eigenvectors of a `n`-by-`n` real symmetric tridiagonal `T` or full symmetric matrix `MAT` (eventually packed column-wise in a linear array *MATP*) corresponding to specified eigenvalues, using inverse iteration and Fernando's method [Golub_VanLoan:1996] [Ipsen:1997] [Fernando:1997] [Bini_etal:2005].

If eigenvectors of a full symmetric matrix `MAT` are wanted, it is required that the original symmetric matrix `MAT` has been reduced to symmetric tridiagonal form `T` by an orthogonal similarity transformation:

$$Q^T * MAT * Q = T$$

with a call to *symtrid_cmp()* with parameter *STORE_Q* set to `true`, before calling **trid_inviter**().

The eigenvectors of `T` are computed using inverse iteration, combined with Fernando's method, for all the eigenvalues at one step. The eigenvectors are then orthogonalized by the Modified Gram-Schmidt or a fast "BLAS3" QR algorithm for large clusters of eigenvalues if the eigenvalues are not well-separated [Golub_VanLoan:1996] [Ipsen:1997].

If eigenvectors of a full symmetric matrix `MAT` are wanted, the computed eigenvectors of `T` are backed-transformed to the eigenvectors of `MAT` using the packed representation of `Q` as computed by *symtrid_cmp()*.

Both the computation of the eigenvectors of `T` and the back-transformation phase are parallelized with OpenMP [Walker:1988].

trid_inviter() may fail if some the eigenvalues specified in parameter *EIGVAL* are nearly identical for some pathological matrices.

Synopsis:

```
call trid_inviter( d(:n) , e(:n) , eigval     , eigvec(:n)    , failure ,     ␣
↪                 maxiter=maxiter , scaling=scaling , initvec=initvec )
call trid_inviter( d(:n) , e(:n) , eigval(:p) , eigvec(:n,:p) , failure ,    ␣
↪                  maxiter=maxiter , ortho=ortho ,  backward_sweep=backward_
↪sweep , scaling=scaling , initvec=initvec )
call trid_inviter( d(:n) , e(:n) , eigval(:p) , eigvec(:n,:p) , failure␣
↪, mat(:n,:n)          , maxiter=maxiter , ortho=ortho ,  backward_
↪sweep=backward_sweep , scaling=scaling , initvec=initvec )
call trid_inviter( d(:n) , e(:n) , eigval(:p) , eigvec(:n,:p) , failure␣
↪, matp(:(n*(n+1)/2)) , maxiter=maxiter , ortho=ortho ,  backward_
↪sweep=backward_sweep , scaling=scaling , initvec=initvec )
```

Examples:

ex1_trid_inviter.F90

ex1_trid_inviter_bis.F90

ex2_trid_inviter.F90

ex3_trid_inviter.F90

gen_symtrid_mat()

Purpose:

gen_symtrid_mat() generates different types of symmetric tridiagonal matrices with known eigenvalues or specific numerical properties such as clustered eigenvalues and so on for testing purposes of eigensolvers.

Optionally, the eigenvalues of the selected symmetric tridiagonal matrix can be computed analytically, if possible, or by a bisection algorithm with high absolute and relative accuracies.

Synopsis:

```
call gen_symtrid_mat( type , d(:n) , e(:n) , failure=failure , known_
↪eigval=known_eigval , eigval=eigval , sort=sort , val1=val1 , val2=val2 ,
↪ l0=l0 , glu0=glu0 )
```

5.20 MODULE SVD_Procedures

Module *SVD_Procedures* exports a large set of routines for computing the full or partial Singular Value Decomposition (SVD), the full or partial QLP Decomposition and the generalized inverse of a matrix and related computations (e.g., bidiagonal reduction of a general matrix, bidiagonal SVD solvers, ...) [Lawson_Hanson:1974] [Golub_VanLoan:1996] [Stewart:1999b] [Reinsch_Richter:2023].

Fast methods for obtaining approximations of a truncated SVD or QLP decomposition of rectangular matrices or EigenValue Decomposition (EVD) of symmetric positive semi-definite matrices based on randomization algorithms are also included [Halko_etal:2011] [Martinsson:2019] [Xiao_etal:2017] [Feng_etal:2019] [Duersch_Gu:2020] [Wu_Xiang:2020].

A general rectangular `m`-by-`n` matrix `MAT` has a SVD into the product of a `m`-by-`min(m,n)` orthogonal matrix `U` (e.g., $U^T * U = I$), a `min(m,n)`-by-`min(m,n)` diagonal matrix of singular values `S` and the transpose of a `n`-by-`min(m,n)` orthogonal matrix `V` (e.g., $V^T * V = I$),

$$MAT = U * S * V^T$$

The singular values $S(i,i) = \sigma_i$ are all non-negative and can be chosen to form a non-increasing sequence,

$$\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_{min(m,n)} \geq 0$$

Note that the driver routines in the *SVD_Procedures* module compute the *thin* version of the SVD with `U` and `V` as `m`-by-`min(m,n)` and `n`-by-`min(m,n)` matrices with orthonormal columns, respectively. This reduces the needed workspace or allows in-place computations and is the most commonly-used SVD form in practice.

Mathematically, the *full* SVD is defined with `U` as a `m`-by-`m` orthogonal matrix, `V` a a `n`-by-`n` orthogonal matrix and `S` as a `m`-by-`n` diagonal matrix (with additional rows or columns of zeros). This *full* SVD can also be computed with the help of the computational routines included in the *SVD_Procedures* module at the user option.

The SVD of a matrix has many practical uses [Lawson_Hanson:1974] [Golub_VanLoan:1996] [Hansen_etal:2012]. The condition number of the matrix (induced by the vector Euclidean norm), if not infinite, is given by the ratio of the largest singular value to the smallest singular value [Golub_VanLoan:1996] and the presence of a zero singular value indicates that the matrix is singular and that this condition number is infinite. The number of non-zero singular values indicates the rank of the matrix. In practice, the SVD of a rank-deficient matrix will not produce exact zeroes for singular values, due to finite numerical precision. Small singular values should be set to zero explicitly by choosing a suitable tolerance and this is the strategy followed for computing the generalized (e.g., Moore-Penrose) inverse MAT^+ of a matrix or for solving rank-deficient linear least squares problems with the SVD [Golub_VanLoan:1996] [Hansen_etal:2012]. See the documentations of *comp_ginv()* subroutine in this module or of *llsq_svd_solve()* and *llsq_svd_solve2()* subroutines in *LLSQ_Procedures* module for more details.

For a rank-deficient matrix, the null space of `MAT` is given by the span of the columns of `V` corresponding to the zero singular values in the *full* SVD of `MAT`. Similarly, the range of `MAT` is given by the span of the columns of `U` corresponding to the non-zero singular values.

See [Lawson_Hanson:1974], [Golub_VanLoan:1996] or [Hansen_etal:2012] for more details on these various results related to the SVD of a matrix.

For very large matrices, the classical SVD algorithms are prohibitively computationally expensive. In such cases, the QLP decomposition provides a reasonable and cheap estimate of the SVD of a matrix, especially when this matrix has a low rank or a significant gap in its singular values spectrum [Stewart:1999b] [Huckaby_Chan:2003] [Huckaby_Chan:2005]. The full or partial QLP decomposition has the form:

$$MAT \simeq Q * L * P^T$$

where `Q` and `P` are `m`-by-`k` and `n`-by-`k` matrices with orthonormal columns (and `k<=min(m,n)`) and `L` is a `k`-by-`k` lower triangular matrix. If `k=min(m,n)`, the QLP factorization is complete and

$$MAT = Q * L * P^T$$

The QLP factorization can be obtained by a two-step algorithm:

- in a first step, a partial (or complete) QR factorization with Column Pivoting (QRCP) of `MAT` is computed;
- in a second step, a LQ decomposition of the (eventually row-permuted in decreasing order of magnitude of the rows) upper triangular or trapezoidal (e.g., if `n`> `m`) factor, `R`, of this QR decomposition is computed, giving the final (pivoted) QLP decomposition.

See the manual of the *QR_Procedures* module for an introduction to the QR, QRCP and LQ factorizations. In many cases, the diagonal elements of `L` track the singular values (in non-increasing order) of `MAT` with a reasonable accuracy [Stewart:1999b] [Huckaby_Chan:2003] [Huckaby_Chan:2005]. This property and the fact that the QRCP and LQ factorizations can be interleaved and stopped at any point where there is a gap in the diagonal elements of `L` make the QLP decomposition a reasonable candidate to determine the rank of a matrix or to obtain a good low-rank approximation of a matrix [Stewart:1999b] [Huckaby_Chan:2003].

Most of the available SVD algorithms in STATPACK and elsewhere split also in two parts. The first part is a reduction of `MAT` to bidiagonal form `BD` in order to reduce the overall computational cost. The second part is the computation of the SVD of this bidiagonal matrix `BD`, which can be accomplished by several different methods [Golub_VanLoan:1996] [Dongarra_etal:2018].

Thus, as intermediate steps for computing the SVD or for obtaining an accurate partial SVD at a reduced cost, this module also provides routines for

- the transformation of `MAT` to bidiagonal form `BD` by similarity transformations [Lawson_Hanson:1974] [Golub_VanLoan:1996] [Dongarra_etal:2018],

  $$MAT = Q * BD * P^T$$

  where `Q` and `P` are orthogonal matrices and `BD` is a `min(m,n)`-by-`min(m,n)` upper or lower bidiagonal matrix (e.g., with non-zero entries only on the diagonal and superdiagonal or on the diagonal and subdiagonal). The shape of `Q` is `m`-by-`min(m,n)` and the shape of `P` is `n`-by-`min(m,n)`.

- the computation of the singular values σ_i, left singular vectors w_i and right singular vectors z_i of `BD`,

  $$W^T * BD * Z = S$$

  where `S` is a `min(m,n)`-by-`min(m,n)` diagonal matrix with $S(i,i) = \sigma_i$, `W` is the `min(m,n)`-by-`min(m,n)` matrix of left singular vectors of `BD` and `Z` is the `min(m,n)`-by-`min(m,n)` matrix of right singular vectors of `BD` stored column-wise;

- the back-transformation of the singular vectors w_i and z_i of `BD` to the singular vectors u_i and v_i of `MAT` [Golub_VanLoan:1996] [Dongarra_etal:2018],

  $$MAT = (Q * W) * S * (P * Z)^T = U * S * V^T$$

where U is the m-by-min(m,n) matrix of the left singular vectors of MAT, V is the n-by-min(m,n) matrix of the right singular vectors of MAT (both stored column-wise) and the singular values $S(i,i) = \sigma_i$ of BD are also the singular values of MAT.

Depending on the situation and the algorithm used, it is also possible to compute only the largest singular values and associated singular vectors of BD and, finally those of MAT.

In what follows, we give some important details about algorithms used in STATPACK for these different intermediate steps: reduction to bidiagonal form BD, computation of the singular values and vectors of BD and back-transformation of the singular vectors of BD to those of MAT.

First, STATPACK includes two different algorithms for the reduction of a matrix to bidiagonal form:

- a cache-efficient blocked and parallel version of the classic Golub and Kahan Householder bidiagonalization [Golub_VanLoan:1996] [Reinsch_Richter:2023], which reduces the traffic on the data bus from four reads and two writes per column-row elimination of the bidiagonalization process to one read and one write [Howell_etal:2008];
- a blocked (e.g., "BLAS3") and parallel version of the one-sided Ralha-Barlow bidiagonal reduction algorithm [Ralha:2003] [Barlow_etal:2005] [Bosner_Barlow:2007] , with an eventual partial reorthogonalization based on Gram-Schmidt orthogonalization [Stewart:2007]. This algorithm requires that m>=n and when m<n it is applied to $\mathrm{MAT}^{\mathrm{T}}$ instead. In serial mode (e.g., without OpenMP), it is significantly faster in most cases than the first one as the input matrix is only accessed column-wise (e.g., it is an one-sided algorithm), but is less accurate for the left orthonormal vectors stored column-wise in Q as these vectors are computed by a recurrence relationship, which can result in a loss of orthogonality for some matrices with a large condition number [Ralha:2003] [Barlow_etal:2005] [Bosner_Barlow:2007]. A partial reorthogonalization procedure based on Gram-Schmidt orthogonalization [Stewart:2007] has been incorporated into the algorithm in order to correct partially this loss of orthogonality, but is not always sufficient, especially for (large) matrices with a slow decay of singular values near zero. Thus, the loss of orthogonality of Q can be severe for these particular matrices. Fortunately, when the one-sided Ralha-Barlow bidiagonal reduction algorithm is used for computing the SVD of a rectangular matrix, the loss of orthogonality concerns only the left singular vectors associated with the smallest singular values [Barlow_etal:2005]. Furthermore, this deficiency can also be easily corrected a posteriori, if needed, giving fully accurate and fast SVDs for any matrix (see the manuals of the *svd_cmp4()*, *svd_cmp5()* and *svd_cmp6()* SVD drivers in this module, which incorporate such a modification, for details).

The first bidiagonalization reduction algorithm is implemented in *bd_cmp()* subroutine and the one-sided Ralha-Barlow bidiagonal reduction algorithm is implemented with partial reorthogonalization in *bd_cmp2()* subroutine and without partial reorthogonalization in *bd_cmp3()* subroutine. Thus, *bd_cmp3()* will be significantly faster than *bd_cmp2()* for low-rank deficient matrices, but a drawback is that Q is not output by *bd_cmp3()* as this matrix will not be numerically orthogonal in some difficult cases. Among the eight general SVD drivers available in STATPACK:

- *svd_cmp()*, *svd_cmp2()* and *svd_cmp7()* use *bd_cmp()* subroutine for the reduction to bidiagonal form of a rectangular matrix;
- *svd_cmp3()* and *svd_cmp8()* use *bd_cmp2()*, e.g., the one-sided Ralha-Barlow bidiagonal reduction algorithm with partial reorthogonalization for this task;
- and, finally, *svd_cmp4()*, *svd_cmp5()* and *svd_cmp6()* use *bd_cmp3()*, e.g., the one-sided Ralha-Barlow bidiagonal reduction algorithm without partial reorthogonalization and a final additional step is required in these SVD drivers to recover all the factors in the SVD of a rectangular matrix.

As a general rule, in a parallel setting (e.g., when OpenMP is used), keep also in mind that the SVD drivers using the *bd_cmp()* subroutine for the reduction to bidiagonal form will be faster if the number of processors available for the computations is large (e.g., greater than 24 or so), as the number of required synchronization barriers is small in *bd_cmp()* subroutine. On the other hand, if the number of available processors is small (e.g., less than 16 or so), the SVD drivers using the *bd_cmp2()* or *bd_cmp3()* bidiagonal reduction subroutines will be generally faster because these routines use blocked (e.g., "BLAS3") algorithms; but the number of required synchronization barriers is

larger in these two routines, which explains why their performance is degraded compared to *bd_cmp()* when many processors are available.

Blocked (e.g., "BLAS3") and parallel routines are also provided for generation and application of the orthogonal matrices, Q and P associated with the bidiagonalization process in both cases [Dongarra_etal:1989] [Golub_VanLoan:1996] [Walker:1988]. See the manuals of *ortho_gen_bd()*, *ortho_gen_bd2()*, *ortho_gen_q_bd()* and *ortho_gen_p_bd()* subroutines for details.

STATPACK also includes several different algorithms for computing all or selected singular values of a bidiagonal matrix BD:

- bidiagonal QR iteration with implicit (Wilkinson) shift [Lawson_Hanson:1974] [Golub_VanLoan:1996] [Reinsch_Richter:2023] ;
- bisection applied to the Tridiagonal Golub-Kahan (TGK) form of a bidiagonal matrix or implicitly to the associated symmetric tridiagonal matrix $BD^T * BD$ [Golub_VanLoan:1996] [Fernando:1998] ;
- and, finally, Differential Quotient Difference with Shifts (DQDS) algorithms [Fernando_Parlett:1994] [Li_etal:2014].

The bidiagonal implicit QR algorithm applies a sequence of similarity transformations to the bidiagonal matrix BD until its off-diagonal elements become negligible and the diagonal elements have converged to the singular values of BD. It consists of a bulge-chasing procedure that implicitly includes shifts and use plane rotations (e.g., Givens rotations) which preserve the bidiagonal form of BD [Lawson_Hanson:1974] [Golub_VanLoan:1996] [Reinsch_Richter:2023]. Bidiagonal QR iteration with implicit shift is implemented in the generic *bd_svd()* and *bd_svd2()* subroutines.

Bisection is a standard method for computing eigenvalues or singular values of a matrix [Golub_VanLoan:1996]. Bisection is based on Sturm sequences and requires $O(min(n,m).k)$ or $O(2min(n,m).k)$ operations to compute k singular values of a min(n,m)-by-min(n,m) bidiagonal matrix BD [Golub_VanLoan:1996] [Fernando:1998]. Two parallel bisection algorithms for bidiagonal matrices are currently implemented in STATPACK:

- The first applies bisection to an associated 2.min(n,m)-by-2.min(n,m) symmetric tridiagonal matrix T with zeros on the diagonal (the so-called Tridiagonal Golub-Kahan form of BD) whose eigenvalues are the singular values of BD and their negatives [Fernando:1998] [Marques_etal:2020]. This approach is implemented in *bd_singval()* subroutine;
- The second applies bisection implicitly to the associated min(n,m)-by-min(n,m) symmetric tridiagonal matrix $BD^T * BD$ (but without computing this matrix product) whose eigenvalues are the squares of the singular values of BD by using the differential stationary form of the QD algorithm of Rutishauser (see Sec.3.1 of [Fernando:1998]). This approach is implemented in *bd_singval2()* subroutine.

If high relative accuracy for small singular values is required, the first bisection algorithm based on the Tridiagonal Golub-Kahan (TGK) form of the bidiagonal matrix is the best choice [Fernando:1998]. Importantly, both STATPACK bidiagonal bisection routines (e.g., *bd_singval()* and *bd_singval2()*) also allow subset computations of the (largest) singular values of BD.

The DQDS algorithm is mathematically equivalent to the Cholesky LR method with shifts, applied to tridiagonal symmetric matrices [Fernando_Parlett:1994]. The DQDS algorithm has been popular due to its accuracy, speed and numerical stability, and is implemented as subroutine dlasq() in LAPACK [Parlett_Marques:2000] [Li_etal:2014]. The first version of the DQDS algorithm included in STATPACK is adapted from this dlasq() subroutine in LAPACK and also includes the early aggressive deflation strategy proposed by [Nakatsukasa_etal:2012] at the user option. The second version of the DQDS algorithm included in STATPACK uses a different shift strategy proposed in [Yamashita_etal:2013]. This variant is usually much slower than the first, but may be more accurate for some bidiagonal matrices. These two variants of the DQDS algorithm are provided in the *bd_lasq1()* and *bd_dqds()* subroutines, respectively.

All these different singular values computational subroutines are available in the *SVD_Procedures* module.

The bidiagonal QR and DQDS methods always compute all the singular values of the bidiagonal matrix BD. On the other hand, subset computations are possible with the two bisection algorithms. Keep also in mind that with both

the bisection and DQDS methods, the singular values can be computed to high relative accuracy, in the absence of denormalization, underflow and overflow, and this property is important if singular vectors need to be computed in a later step by inverse iteration or deflation (see below). Finally, note that the DQDS algorithm is usually much faster than bisection and as fast than QR if all singular values of `BD` must be computed and despite that the STATPACK bisection algorithms are parallelized when OpenMP is used.

Among the eight general SVD drivers available in STATPACK:

- *svd_cmp()*, *svd_cmp2()*, *svd_cmp3()*, *svd_cmp4()* and *svd_cmp5()* use the bidiagonal QR iteration with implicit shift method when both all the singular values and vectors of `BD` are requested and the DQDS algorithm (by default) when only singular values of the input matrix `MAT` are requested;
- *svd_cmp6()* uses bisection on the TGK form of `BD` for computing all or the leading singular values of `BD` (and the input matrix `MAT`) in all cases;
- and, finally, *svd_cmp7()* and *svd_cmp8()* compute all the singular values of `BD` (and the input matrix `MAT`) by the first variant of the DQDS algorithm (e.g., *bd_lasq1()*).

Currently, STATPACK also includes three different methods for computing (selected) singular vectors of a bidiagonal matrix `BD`:

- bidiagonal QR iteration with implicit (Wilkinson) shift [Lawson_Hanson:1974] [Golub_VanLoan:1996] [Reinsch_Richter:2023];
- inverse iteration on the Tridiagonal Golub-Kahan (TGK) form of a bidiagonal matrix [Godunov_etal:1993] [Ipsen:1997] [Dhillon:1998] [Marques_Vasconcelos:2017] [Marques_etal:2020];
- and a novel deflation perfect shift technique applied directly to the bidiagonal matrix `BD` based on the works of Godunov and coworkers on the deflation for tridiagonal matrices [Godunov_etal:1993] [Fernando:1997] and [Malyshev:2000].

The bidiagonal QR algorithm is considered as very accurate and robust for computing singular values [Demmel_Kahan:1990], but its implementation in current Dense Linear Algebra softwares for computing singular vectors is far from optimal in terms of speed [VanZee_etal:2011] [Dongarra_etal:2018]. To correct this deficiency and get high performance for computing singular vectors by this bidiagonal implicit QR algorithm, several modifications have been introduced in its implementation in STATPACK by:

- restructuring the bidiagonal QR iterations with a wave-front algorithm for the accumulation of Givens rotations [VanZee_etal:2011];
- the use of a blocked "BLAS3" algorithm to update the singular vectors by these Givens rotations when possible [Lang:1998];
- the use of a novel perfect shift strategy in the QR iterations inspired by the works of [Godunov_etal:1993] [Malyshev:2000] and [Fernando:1997], which reduces significantly the number of QR iterations needed for convergence for many matrices;
- and, finally, OpenMP parallelization [Demmel_etal:1993].

With all these changes, the bidiagonal QR algorithm becomes competitive with the divide-and-conquer method for computing the *full* or *thin* SVD of a matrix [VanZee_etal:2011]. Subroutines *bd_svd()* and *bd_svd2()* use this efficient and improved bidiagonal implicit QR algorithm. However, a drawback is that subset computations are not possible with the bidiagonal QR algorithm [Marques_etal:2020]. With this method, it is possible to compute all the singular values or both all the singular values and associated singular vectors.

The bisection-inverse iteration or bisection-deflation methods are the preferred methods if you are only interested in a subset of the singular vectors and values of a bidiagonal matrix `BD` or a full matrix `MAT` [Marques_etal:2020].

Once singular values have been obtained by bisection, DQDS or implicit QR iterations in a first step, associated singular vectors can be computed efficiently using:

- Fernando's method and inverse iteration on the TGK form of the bidiagonal matrix `BD` [Godunov_etal:1993] [Fernando:1997] [Bini_etal:2005] [Marques_Vasconcelos:2017] [Marques_etal:2020]. These singular vectors are then orthogonalized by the modified Gram-Schmidt or QR algorithms if the singular values are not well-separated. A "BLAS3" and parallelized QR algorithm is used for large clusters of singular values for increased efficiency;
- a novel technique combining an extension to bidiagonal matrices of Fernando's approach for computing eigenvectors of tridiagonal matrices with a deflation procedure by Givens rotations originally developed by Godunov and collaborators [Fernando:1997] [Parlett_Dhillon:1997] [Malyshev:2000]. If this deflation technique failed, QR bidiagonal iterations with a perfect shift strategy are used instead as a back-up procedure [Mastronardi_etal:2006]. It is highly recommended to compute the singular values of the bidiagonal matrix `BD` to high accuracy for the success of the deflation technique, meaning that this approach is less robust than the inverse iteration technique for computing selected singular vectors of a bidiagonal matrix.

If the distance between the singular values of `BD` is sufficient relative to the norm of `BD`, then computing the associated singular vectors by inverse iteration or deflation is also a $O(min(n,m).k)$ or $O(2.min(n,m).k)$ process, where `k` is the number of singular vectors to compute. Thus, when all singular values are *well* separated and no orthogonalization is needed, the inverse iteration or deflation methods are much faster than than the bidiagonal implicit QR algorithm for computing the singular vectors in the *thin* SVD of a matrix. Furthermore, as already discussed above, the inverse iteration or deflation methods are the preferred methods if you are only interested in a subset of the singular vectors of the matrix `BD` or `MAT`, as subset computations are not possible in the standard implicit QR algorithm.

`bd_inviter()` and *`bd_deflate()`* generic subroutines implement, respectively, the inverse iteration and deflation methods for computing all or selected singular vectors of bidiagonal matrices and *`bd_inviter2()`* and *`bd_deflate2()`* generic subroutines perform the same tasks, but for full matrices, once these full matrices have been reduced to bidiagonal form and singular values have been computed by bisection or DQDS. Note that these preliminary tasks can be done efficiently with the help of the *`select_singval_cmp()`*, *`select_singval_cmp2()`*, *`select_singval_cmp3()`* and *`select_singval_cmp4()`* subroutines also available in this module.

Among the eight general SVD drivers available in STATPACK, *`svd_cmp6()`*, *`svd_cmp7()`* and *`svd_cmp8()`* uses the inverse iteration method for computing all or the leading singular vectors of `BD` (and the input matrix `MAT`).

All above algorithms are parallelized with OpenMP [openmp]. Parallelism concerns only the computation of singular vectors in the bidiagonal implicit QR method, but both the computation of the singular values and singular vectors in the bisection-inverse iteration and bisection-deflation methods. Furthermore, the computation of singular vectors in the bidiagonal QR or inverse iteration methods also uses blocked "BLAS3" algorithms when possible for maximum efficiency [Walker:1988] [Lang:1998].

Note finally that the SVD driver and computational routines provided in this module are very different from the corresponding bidiagonal implicit QR iteration and inverse iteration routines provided by LAPACK [Anderson_etal:1999] and are much faster if OpenMP and BLAS supports are activated, but slightly less accurate for the same precision in their default settings for a few cases.

In addition to these *standard* and *deterministic* SVD (or QLP) driver and computational routines based on bidiagonal QR, inverse or deflation iterations applied to bidiagonal matrices after a preliminary bidiagonal reduction step, module *SVD_Procedures* also includes an extensive set of very fast routines based on randomization techniques for solving low-rank matrix approximation problems with a much better efficiency (but a decreased accuracy). These new methods are becoming increasingly important as recent applications require larger and larger datasets that the traditional methods described above cannot adequately handle in a reasonable computing time.

For a good and general introduction to randomized linear algebra, see [Li_etal:2017], [Martinsson:2019] and [Erichson_etal:2019]. The recent survey [Tropp_Webber:2023], which focuses specifically on modern approaches for computing low-rank approximations of high-dimensional matrices by means of the randomized SVD, randomized subspace iteration, and randomized block Krylov iteration, is also an excellent reference for many of the randomized algorithms included in module *SVD_Procedures* for obtaining a low-rank approximation of a high-dimensional matrix at a reduced cost. Typically, these low-rank methods approximate an input matrix as a product of several smaller matrices and can be used to manipulate efficiently this input matrix at reduced costs. There are two classes of randomized

low-rank approximation algorithms, sampling-based and random projection-based algorithms:

- Sampling algorithms use randomly selected columns or rows based on sampling probabilities derived from the original matrix in a first step, and a deterministic algorithm, such as SVD or EVD, is performed on the smaller subsampled matrix;
- the projection-based algorithms use the concept of random projections to project the high-dimensional space spanned by the columns of the matrix into a low-dimensional subspace, which approximates the dominant subspace of a matrix. The input matrix is then compressed-either explicitly or implicitly-to this subspace, and the reduced matrix is manipulated inexpensively by *standard* deterministic methods to obtain the desired low-rank factorization in a second step.

The randomized routines included in module *SVD_Procedures* are projection-based methods. In many cases, this approach beats largely its classical competitors in terms of speed [Halko_etal:2011] [Musco_Musco:2015] [Li_etal:2017] [Tropp_Webber:2023]. Thus, these routines based on recent randomized projection algorithms are much faster than the *standard* drivers included in module *SVD_Procedures* or *Eig_Procedures* for computing a truncated SVD or EVD of a matrix. Yet, such randomized methods are also shown to compute with a very high probability low-rank approximations that are accurate, and are known to perform even better in many practical situations when the singular values of the input matrix decay quickly [Halko_etal:2011] [Gu:2015] [Li_etal:2017] [Tropp_Webber:2023].

More precisely, module *SVD_Procedures* includes routines based on randomization for computing:

- approximations of the largest singular values and associated left and right singular vectors of full general matrices using randomized power, subspace or block Krylov iterations [Halko_etal:2011] [Gu:2015] [Musco_Musco:2015] [Li_etal:2017] [Tropp_Webber:2023] or randomized QRCP and QLP factorizations [Duersch_Gu:2017] [Xiao_etal:2017] [Feng_etal:2019] [Duersch_Gu:2020] in order to extract the dominant subspace of a matrix;
- approximations of the largest eigenvalues and associated eigenvectors of full symmetric positive semi-definite matrices using randomized power, subspace or block Krylov iterations or the Nystrom method [Halko_etal:2011] [Musco_Musco:2015] [Li_etal:2017]. The Nystrom method provides more accurate results for positive semi-definite matrices;
- and, finally, randomized partial QLP factorizations, which are also much faster than their deterministic counterparts [Wu_Xiang:2020];

Usually, the problem of low-rank matrix approximation falls into two categories:

- the fixed-rank problem, where the rank `nsvd` of the matrix approximation which is sought is given;
- the fixed-precision problem, where we seek a partial SVD factorization, `rSVD`, with a rank as small as possible, such that

$$||MAT - rSVD||_F <= eps$$

  where `eps` is a given accuracy tolerance. Algorithms for solving the fixed-precision problem that depend on a specific tolerance are important in practice as, in most cases, the users don't know the proper rank of the matrix approximation to use beforehand.

Module *SVD_Procedures* includes four (randomized) routines for solving the fixed-rank problem: *rqr_svd_cmp()*, *rsvd_cmp()*, *rqlp_svd_cmp()* and *rqlp_svd_cmp2()* and three (randomized) routines for solving the fixed-precision problem: *rqr_svd_cmp_fixed_precision()*, *rsvd_cmp_fixed_precision()*, *rqlp_svd_cmp_fixed_precision()*.

rqr_svd_cmp() and *rqr_svd_cmp_fixed_precision()* are based on a (randomized or deterministic) partial QRCP factorization followed by a SVD step [Xiao_etal:2017]. *rsvd_cmp()* and *rsvd_cmp_fixed_precision()* are based on randomized power, subspace or block Krylov iterations followed by a SVD step [Musco_Musco:2015] [Li_etal:2017] [Martinsson_Voronin:2016] [Yu_etal:2018] and, finally, *rqlp_svd_cmp()*, *rqlp_svd_cmp2()* and *rqlp_svd_cmp_fixed_precision()* are based on a (randomized or deterministic) partial QLP factorization followed by a SVD step [Duersch_Gu:2017] [Feng_etal:2019] [Duersch_Gu:2020].

The choice between these different subroutines involves tradeoffs between speed and accuracy. For both the fixed-rank and fixed-precision problems, the routines based on a preliminary partial QRCP factorization are the fastest and the less accurate, and those based on a partial QLP factorization are the most accurate, but the slowest. On the other hand, routines based on randomized power, subspace or block Krylov iterations provide intermediate performances both in terms of accuracy and speed in most cases. In this class of methods, routines based on randomized power iteration return accurate approximations for matrices with rapid singular value decay, but algorithms based on randomized block Krylov iteration perform better for matrices with a moderate or slow singular value decay [Tropp_Webber:2023]. Finally, in case of limited storage, algorithms based on subspace iterations are a good compromise [Tropp_Webber:2023]. Keep also in mind, that if you already know the rank of the matrix approximation you are looking for, routines dedicated to solve the fixed-rank problem (.e.g., *rqr_svd_cmp()*, *rsvd_cmp()*, *rqlp_svd_cmp()* and *rqlp_svd_cmp2()*) are faster and more accurate than their twin routines dedicated to solve the fixed-precision problem.

Module *SVD_Procedures* also includes a subroutine, *reig_pos_cmp()*, for computing approximations of the largest eigenvalues and associated eigenvectors of a full n-by-n real symmetric positive semi-definite matrix MAT using a variety of randomized techniques and, also, three subroutines, *qlp_cmp()*, *qlp_cmp2()* and *rqlp_cmp()*, for computing randomized (or deterministic) full or partial QLP factorizations, which can also be used to solve the fixed-rank problem [Stewart:1999b].

All these randomized SVD or QLP algorithms are also parallelized with OpenMP [openmp].

Finally, note that the routines provided in this module apply only to real data of kind **stnd**. The real kind type **stnd** is defined in module *Select_Parameters*. Computation of singular values and vectors for a complex matrix are not provided in this release of STATPACK.

In order to use one of these routines, you must include an appropriate `use SVD_Procedures` or `use Statpack` statement in your Fortran program, like:

```
use SVD_Procedures, only: svd_cmp
```

or :

```
use Statpack, only: svd_cmp
```

Here is the list of the public routines exported by module *SVD_Procedures*:

bd_cmp()

Purpose:

bd_cmp() reduces a general m-by-n matrix MAT to upper or lower bidiagonal form BD by an orthogonal transformation:

$$Q^T * MAT * P = BD$$

where Q and P are orthogonal matrices. If:

- m >= n, BD is upper bidiagonal;
- m < n, BD is lower bidiagonal.

bd_cmp() computes BD, Q and P, using an efficient variant of the classic Golub and Kahan Householder bidiagonalization algorithm [Howell_etal:2008].

Optionally, **bd_cmp()** can also reduce a general m-by-n matrix MAT to upper bidiagonal form BD by a two-step algorithm:

- If m >= n, a QR factorization of the real m-by-n matrix MAT is first computed

  $$MAT = O * R$$

  where O is orthogonal and R is upper triangular. In a second step, the n-by-n upper triangular matrix R is reduced to upper bidiagonal form BD by an orthogonal transformation:

$$Q^T * R * P = BD$$

where Q and P are orthogonal and BD is an upper bidiagonal matrix.

- If m < n, a LQ factorization of the real m-by-n matrix MAT is first computed

  $$MAT = L * O$$

  where O is orthogonal and L is lower triangular. In a second step, the m-by-m lower triangular matrix L is reduced to upper bidiagonal form BD by an orthogonal transformation :

  $$Q^T * L * P = BD$$

  where Q and P are orthogonal and BD is an upper bidiagonal matrix.

This two-step reduction algorithm will be more efficient if m is much larger than n or if n is much larger than m.

These two different reduction algorithms of MAT to bidiagonal form BD are also parallelized with OpenMP.

Synopsis:

```
call bd_cmp( mat(:m,:n) , d(:min(m,n)) , e(:min(m,n)) , tauq(:min(m,n)) ,␣
↪taup(:min(m,n)) )

call bd_cmp( mat(:m,:n) , d(:min(m,n)) , e(:min(m,n)) , tauq(:min(m,n))       ␣
↪             )

call bd_cmp( mat(:m,:n) , d(:min(m,n)) , e(:min(m,n))                         ␣
↪             )

call bd_cmp( mat(:m,:n) , d(:min(m,n)) , e(:min(m,n)) , tauq(:min(m,n)) ,␣
↪taup(:min(m,n)), rlmat(:min(m,n),:min(m,n)) , tauo=tauo(:min(m,n)) )
```

Examples:

ex1_bd_cmp.F90

ex1_apply_q_bd.F90

ex1_bd_inviter2.F90

ex1_bd_inviter2_bis.F90

ex1_bd_deflate2.F90

ex1_bd_deflate2_bis.F90

bd_cmp2 ()

Purpose:

bd_cmp2() reduces a m-by-n matrix MAT with m >= n to upper bidiagonal form BD by an orthogonal transformation:

$$Q^T * MAT * P = BD$$

where Q and P are orthogonal matrices.

bd_cmp2() computes BD, Q and P using a parallel (if OpenMP support is activated) and blocked version of the one-sided Ralha-Barlow bidiagonal reduction algorithm [Ralha:2003] [Barlow_etal:2005] [Bosner_Barlow:2007].

A partial reorthogonalization procedure based on Gram-Schmidt orthogonalization [Stewart:2007] has been incorporated to **bd_cmp2()** in order to correct partially the loss of orthogonality of Q in the one-sided Ralha-Barlow bidiagonal reduction algorithm [Barlow_etal:2005].

bd_cmp2() is more efficient than *bd_cmp()* as it is an one-sided algorithm, but is less accurate for the computation of the orthogonal matrix Q.

Synopsis:

```
call bd_cmp2( mat(:m,:n) , d(:n) , e(:n) , p(:n,:n), failure=failure , gen_
↪p=gen_p, reortho=reortho )
call bd_cmp2( mat(:m,:n) , d(:n) , e(:n) ,           failure=failure
↪     , reortho=reortho )
```

Examples:

ex1_bd_cmp2.F90

ex2_bd_inviter2.F90

ex2_bd_deflate2.F90

bd_cmp3()

Purpose:

bd_cmp3() reduces a m-by-n matrix MAT with m >= n to upper bidiagonal form BD by an orthogonal transformation:

$$Q^T * MAT * P = BD$$

where Q and P are orthogonal.

bd_cmp3() computes BD and P using a parallel (if OpenMP support is activated) and blocked version of the one-sided Ralha-Barlow bidiagonal reduction algorithm [Ralha:2003] [Barlow_etal:2005] [Bosner_Barlow:2007].

bd_cmp3() is faster than *bd_cmp2()*, if the Q orthogonal matrix is not needed, especially for matrices with a low rank compared to its dimensions as partial reorthogonalization of Q is not performed in **bd_cmp3**().

Synopsis:

```
call bd_cmp3( mat(:m,:n) , d(:n) , e(:n) , gen_p=gen_p , failure=failure )
```

Examples:

ex1_bd_cmp3.F90

ortho_gen_bd()

Purpose:

ortho_gen_bd() generates the real orthogonal matrices Q and P determined by *bd_cmp()* when reducing a m-by-n real matrix MAT to bidiagonal form:

$$MAT = Q * BD * P^T$$

Q and P are defined as products of elementary reflectors H(i) and G(i), respectively, as computed by *bd_cmp()* and stored in its array arguments *MAT*, *TAUQ* and *TAUP*.

If m >= n:

- $Q = H(1) * H(2) * ... * H(n)$ and **ortho_gen_bd**() returns the first n columns of Q in *MAT*;
- $P = G(1) * G(2) * ... * G(n-1)$ and **ortho_gen_bd**() returns P as an n-by-n matrix in *P*.

If m < n:

- $Q = H(1) * H(2) * ... * H(m-1)$ and **ortho_gen_bd**() returns Q as an m-by-m matrix in *MAT(1:m,1:m)*;
- $P = G(1) * G(2) * ... * G(m)$ and **ortho_gen_bd**() returns the first m columns of P, in *P*.

The generation of the real orthogonal matrices Q and P is blocked and parallelized with OpenMP [Walker:1988].

Synopsis:

```
call ortho_gen_bd( mat(:m,:n) , tauq(:min(m,n)) , taup(:min(m,n)) , p(:n,
↪:min(m,n)) )
```

Examples:

ex1_bd_cmp.F90

ortho_gen_bd2()

Purpose:

ortho_gen_bd2() generates the real orthogonal matrices Q and P^T determined by *bd_cmp()* when reducing a m-by-n real matrix MAT to bidiagonal form:

$$MAT = Q * BD * P^T$$

Q and P^T are defined as products of elementary reflectors H(i) and G(i), respectively, as computed by *bd_cmp()* and stored in its array arguments *MAT*, *TAUQ* and *TAUP*.

If m >= n:

- $Q = H(1) * H(2) * ... * H(n)$ and **ortho_gen_bd2()** returns the first n columns of Q in *MAT*;
- $P^T = G(n-1) * ... * G(2) * G(1)$ and **ortho_gen_bd2()** returns P^T as a n-by-n matrix in *Q_PT*.

If m < n:

- $Q = H(1) * H(2) * ... * H(m-1)$ and **ortho_gen_bd2()** returns Q as a m-by-m matrix in *Q_PT*;
- $P^T = G(m) * ... * G(2) * G(1)$ and **ortho_gen_bd2()** returns the first m rows of P^T, in *MAT*.

The generation of the real orthogonal matrices Q and P^T is blocked and parallelized with OpenMP [Walker:1988].

Synopsis:

```
call ortho_gen_bd2( mat(:m,:n) , tauq(:min(m,n)) , taup(:min(m,n)) , q_
↪pt(:min(m,n),:min(m,n)) )
```

ortho_gen_q_bd()

Purpose:

ortho_gen_q_bd() generate the real orthogonal matrix Q determined by *bd_cmp()* when reducing a m-by-n real matrix MAT to bidiagonal form:

$$MAT = Q * BD * P^T$$

Q is defined as products of elementary reflectors H(i) as computed by *bd_cmp()* and stored in its array arguments *MAT* and *TAUQ*.

If m >= n:

- $Q = H(1) * H(2) * ... * H(n)$ and **ortho_gen_q_bd()** returns the first n columns of Q in *MAT*;

If m < n:

- $Q = H(1) * H(2) * ... * H(m-1)$ and **ortho_gen_q_bd()** returns Q as a m-by-m matrix in *MAT(1:m,1:m)*;

The generation of the real orthogonal matrix Q is blocked and parallelized with OpenMP [Walker:1988].

Synopsis:

```
call ortho_gen_q_bd( mat(:m,:n) , tauq(:min(m,n)) )
```

Examples:

ex1_ortho_gen_q_bd.F90

ortho_gen_p_bd()

Purpose:

ortho_gen_p_bd() generate the real orthogonal matrix `P` determined by *bd_cmp()* when reducing a `m`-by-`n` real matrix `MAT` to bidiagonal form:

$$MAT = Q * BD * P^T$$

`P` is defined as products of elementary reflectors `G(i)` determined by *bd_cmp()* and stored in its array arguments *MAT* and *TAUP*.

If `m` >= `n`:

- $P = G(1) * G(2) * ... * G(n-1)$ and **ortho_gen_p_bd()** returns `P` as a `n`-by-`n` matrix in *P*.

If `m` < `n`:

- $P = G(1) * G(2) * ... * G(m)$ and **ortho_gen_p_bd()** returns the first `m` columns of `P`, in *P*.

The generation of the real orthogonal matrix `P` is blocked and parallelized with OpenMP [Walker:1988].

Synopsis:

```
call ortho_gen_p_bd( mat(:m,:n) , taup(:min(m,n)) , p(:n,:min(m,n)) )
```

Examples:

ex1_ortho_gen_q_bd.F90

apply_q_bd()

Purpose:

apply_q_bd() overwrites the general real `m`-by-`n` matrix `C` with:

- $Q * C$ if *LEFT* = `true` and *TRANS* = `false` ;
- $Q^T * C$ if *LEFT* = `true` and *TRANS* = `true` ;
- $C * Q$ if *LEFT* = `false` and *TRANS* = `false` ;
- $C * Q^T$ if *LEFT* = `false` and *TRANS* = `true` .

Here `Q` is the orthogonal matrix determined by *bd_cmp()* when reducing a real matrix `MAT` to bidiagonal form:

$$MAT = Q * BD * P^T$$

and `Q` is defined as products of elementary reflectors `H(i)` .

Let `nq = m` if *LEFT* = `true` and `nq = n` if *LEFT* = `false`. Thus, `nq` is the order of the orthogonal matrix `Q` that is applied. `MAT` is assumed to have been a `nq`-by-`k` matrix and

$Q = H(1) * H(2) * ... * H(k)$, if `nq >= k`;

or

$Q = H(1) * H(2) * ... * H(nq-1)$, if `nq < k`.

The application of the real orthogonal matrix `Q` to the matrix `C` is blocked and parallelized with OpenMP [Walker:1988].

Synopsis:

```
call apply_q_bd( mat(:m,:n) , tauq(:min(m,n)) , c(:,:) , left , trans )
```

Examples:

ex1_apply_q_bd.F90

ex2_bd_singval.F90

ex2_bd_singval2.F90

apply_p_bd()

Purpose:

apply_p_bd() overwrites the general real m-by-n matrix C with

- $P * C$ if *LEFT* = true and *TRANS* = false ;
- $P^T * C$ if *LEFT* = true and *TRANS* = true ;
- $C * P$ if *LEFT* = false and *TRANS* = false ;
- $C * P^T$ if *LEFT* = false and *TRANS* = true .

Here P is the orthogonal matrix determined by *bd_cmp()* when reducing a real matrix MAT to bidiagonal form:

$$MAT = Q * BD * P^T$$

and P is defined as products of elementary reflectors G(i) .

Let np = m if *LEFT* = true and np = n if *LEFT* = false. Thus, np is the order of the orthogonal matrix P that is applied. MAT is assumed to have been a k-by-np matrix and

$$P = G(1) * G(2) * ... * G(k)$$, if k < np;

or

$$P = G(1) * G(2) * ... * G(np - 1)$$, if k >= np.

The application of the real orthogonal matrix P to the matrix C is blocked and parallelized with OpenMP [Walker:1988].

Synopsis:

```
call apply_p_bd( mat(:m,:n) , taup(:min(m,n)) , c(:,:) , left , trans )
```

Examples:

ex1_apply_q_bd.F90

ex2_bd_singval.F90

ex2_bd_singval2.F90

bd_svd()

Purpose:

bd_svd() computes the Singular Value Decomposition (SVD) of a real n-by-n (upper or lower) bidiagonal matrix B:

$$B = Q * S * P^T$$

where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and, Q and P are orthogonal matrices (P^T denotes the transpose of P).

The routine computes S, U * Q, and V * P, for given real input matrices U, V, using the implicit bidiagonal QR method [Lawson_Hanson:1974] [Golub_VanLoan:1996].

Synopsis:

```
call bd_svd( upper , d(:n) , e(:n) , failure , u(:,:n) , v(:,:n) , sort=sort␣
↪, maxiter=maxiter , max_francis_steps=max_francis_steps , perfect_
↪shift=perfect_shift , bisect=bisect , dqds=dqds )
```

```
call bd_svd( upper , d(:n) , e(:n) , failure , u(:,:n) ,           sort=sort␣
↪, maxiter=maxiter , max_francis_steps=max_francis_steps , perfect_
↪shift=perfect_shift , bisect=bisect , dqds=dqds )

call bd_svd( upper , d(:n) , e(:n) , failure ,                     sort=sort ,
↪ maxiter=maxiter )
```

Examples:

ex1_bd_svd.F90

ex2_bd_svd.F90

ex1_bd_inviter.F90

bd_svd2 ()

Purpose:

bd_svd2() computes the Singular Value Decomposition (SVD) of a real n-by-n (upper or lower) bidiagonal matrix B:

$$B = Q * S * P^T$$

where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and, Q and P are orthogonal matrices (P^T denotes the transpose of P).

The routine computes S, U * Q, and P^T * VT, for given real input matrices U, VT, using the implicit bidiagonal QR method [Lawson_Hanson:1974] [Golub_VanLoan:1996].

Synopsis:

```
call bd_svd2( upper , d(:n) , e(:n) , failure , u(:,:n) , vt(:n,:) ,␣
↪sort=sort , maxiter=maxiter , max_francis_steps=max_francis_steps , perfect_
↪shift=perfect_shift , bisect=bisect , dqds=dqds )

call bd_svd2( upper , d(:n) , e(:n) , failure , u(:,:n) ,          ␣
↪sort=sort , maxiter=maxiter , max_francis_steps=max_francis_steps , perfect_
↪shift=perfect_shift , bisect=bisect , dqds=dqds )

call bd_svd2( upper , d(:n) , e(:n) , failure ,                    ␣
↪sort=sort , maxiter=maxiter )
```

Examples:

ex1_bd_svd2.F90

bd_singval ()

Purpose:

bd_singval() computes all or some of the greatest singular values of a real n-by-n (upper or lower) bidiagonal matrix B by a bisection algorithm.

The Singular Value Decomposition of B is:

$$B = Q * S * P^T$$

where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and, Q and P are orthogonal matrices (P^T denotes the transpose of P).

The singular values S of the bidiagonal matrix B are computed by a bisection algorithm applied to the Tridiagonal Golub-Kahan (TGK) form of the bidiagonal matrix B (see [Fernando:1998]; Sec.3.3).

The singular values can be computed with high relative accuracy, at the user option, by using the optional argument *ABSTOL* with the value `sqrt(lamch("S"))` (which is equal to the square root of the public numerical constant `safmin` exported by the *Num_Constants* module).

Synopsis:

```
call bd_singval( d(:n) , e(:n) , nsing , s(:n) , failure , sort=sort ,␣
↪vector=vector , abstol=abstol , ls=ls , theta=theta , scaling=scaling ,␣
↪init=init )
```

Examples:

ex1_bd_singval.F90

ex1_bd_deflate.F90

ex1_bd_deflate2_bis.F90

ex1_bd_inviter2_bis.F90

ex2_bd_singval.F90

bd_singval2()

Purpose:

bd_singval2() computes all or some of the greatest singular values of a real `n`-by-`n` (upper or lower) bidiagonal matrix `B` by a bisection algorithm.

The Singular Value Decomposition of `B` is:

$$B = Q * S * P^T$$

where `S` is a diagonal matrix with non-negative diagonal elements (the singular values of `B`), and, `Q` and `P` are orthogonal matrices (P^T denotes the transpose of `P`).

The singular values `S` of the bidiagonal matrix `B` are computed by a bisection algorithm (see [Golub_VanLoan:1996]; Sec.8.5). The bisection method is applied (implicitly) to the associated `n`-by-`n` symmetric tridiagonal matrix

$$B^T * B$$

whose eigenvalues are the squares of the singular values of `B` by using the differential stationary form of the qd algorithm of Rutishauser (see [Fernando:1998]; Sec.3.1).

The singular values can be computed with high accuracy, at the user option, by using the optional argument *ABSTOL* with the value `sqrt(lamch("S"))` (which is equal to the square root of the numerical constant `safmin` in the *Num_Constants* module).

Synopsis:

```
call bd_singval2( d(:n) , e(:n) , nsing , s(:n) , failure , sort=sort ,␣
↪vector=vector , abstol=abstol , ls=ls , theta=theta , scaling=scaling ,␣
↪init=init )
```

Examples:

ex1_bd_singval2.F90

ex1_bd_inviter2.F90

ex1_bd_deflate2.F90

ex2_bd_singval2.F90

bd_max_singval()

Purpose:

bd_max_singval() computes the greatest singular value of a real n-by-n (upper or lower) bidiagonal matrix B by a bisection algorithm.

The Singular Value Decomposition of B is:

$$B = Q * S * P^T$$

where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and, Q and P are orthogonal matrices (P^T denotes the transpose of P).

The greatest singular value of the bidiagonal matrix B is computed by a bisection algorithm (see [Golub_VanLoan:1996]; Sec.8.5). The bisection method is applied (implicitly) to the associated n-by-n symmetric tridiagonal matrix

$$B^T * B$$

whose eigenvalues are the squares of the singular values of B by using the differential stationary form of the qd algorithm of Rutishauser (see [Fernando:1998]; Sec.3.1).

The greatest singular value can be computed with high accuracy, at the user option, by using the optional argument *ABSTOL* with the value `sqrt(lamch("S"))` (which is equal to the constant `safmin` in the *Num_Constants* module).

Synopsis:

```
call bd_max_singval( d(:n) , e(:n) , nsing , s(:n) , failure , abstol=abstol ,
↪ scaling=scaling )
```

bd_lasq1()

Purpose:

bd_lasq1() computes all singular values of a real n-by-n (upper or lower) bidiagonal matrix B by a Differential Quotient Difference with Shifts (DQDS) algorithm and, optionally, with an early aggressive deflation strategy [Fernando_Parlett:1994] [Parlett_Marques:2000] [Li_etal:2014] [Nakatsukasa_etal:2012].

The Singular Value Decomposition of B is:

$$B = Q * S * P^T$$

where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and, Q and P are orthogonal matrices (P^T denotes the transpose of P).

The singular values S of the bidiagonal matrix B are computed to high relative accuracy, in the absence of denormalization, underflow and overflow.

This routine is an improved version of subroutine `DLASQ()` in LAPACK [Parlett_Marques:2000] [Li_etal:2014], but also includes the early aggressive deflation strategy described in [Nakatsukasa_etal:2012] for large bidiagonal matrices or at the user option.

Synopsis:

```
call bd_lasq1( d(:n) , e(:n) , failure , maxiter=maxiter , sort=sort ,␣
↪scaling=scaling , ieee=ieee , aggdef2=aggdef2 , max_win=max_win , freq=freq␣
↪, info=info )
```

Examples:

ex1_bd_lasq1.F90

bd_dqds()

Purpose:

bd_dqds() computes all singular values of a real n-by-n (upper or lower) bidiagonal matrix B by a Differential Quotient Difference with Shifts (DQDS) algorithm [Fernando_Parlett:1994] [Yamashita_etal:2013] .

The Singular Value Decomposition of B is:

$$B = Q * S * P^T$$

where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and, Q and P are orthogonal matrices (P^T denotes the transpose of P).

The singular values S of the bidiagonal matrix B are computed to high relative accuracy, in the absence of denormalization, underflow and overflow.

This routine used the more complex shift strategy described in [Yamashita_etal:2013] . **bd_dqds()** is significantly slower than **bd_lasq1()** due to its more complex shift strategy, but can be more accurate for some bidiagonal matrices as the number of DQDS iterations are reduced in **bd_dqds()**.

Synopsis:

```
call bd_dqds( d(:n) , e(:n) , failure , maxiter=maxiter , sort=sort ,␣
↪scaling=scaling )
```

Examples:

ex1_bd_dqds.F90

las2()

Purpose:

las2() computes the two singular values of a real 2-by-2 bidiagonal matrix B.

The Singular Value Decomposition of B is:

$$B = Q * S * P^T$$

where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and, Q and P are orthogonal matrices (P^T denotes the transpose of P).

This routine is adapted from subroutine DLAS2() in LAPACK. Barring over/underflow, all output quantities are correct to within a few units in the last place.

Synopsis:

```
call las2( f , g , h , ssmin , ssmax )
```

svd_cmp()

Purpose:

svd_cmp() computes the Singular Value Decomposition (SVD) of a real m-by-n matrix MAT. The SVD is written:

$$MAT = U * S * V^T$$

where S is a m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is a m-by-m orthogonal matrix, and V is a n-by-n orthogonal matrix. The diagonal elements of S are the singular values of MAT; they are real and non-negative. The columns of U and V are, respectively, the left and right singular vectors of MAT.

svd_cmp() computes only the first min(m,n) columns of U and V (e.g., the left and right singular vectors of MAT in the *thin* SVD of MAT).

The routine returns the min(m,n) singular values and the associated left and right singular vectors. The singulars vectors are returned column-wise in all cases.

The following three-step algorithm is used to compute the *thin* SVD of MAT:

- MAT is first reduced to bidiagonal form B with the help of a (parallel) fast and cache efficient variant of the Golub-Kahan bidiagonal reduction algorithm in a first step [Howell_etal:2008]. Furthermore, if m is much larger than n or if n is much larger than m, a preliminary QR or LQ factorization is performed, respectively, and the Golub-Kahan bidiagonal reduction algorithm is applied to the resulting triangular factor.
- In place accumulation of the left and right orthogonal transformations, used in the reduction of MAT to bidiagonal form B, is performed by fast blocked and parallel algorithms in a second step [Lawson_Hanson:1974] [Walker:1988] [Golub_VanLoan:1996].
- The singular values and vectors of the bidiagonal matrix MAT are then computed by a blocked (e.g., "BLAS3"), cache efficient and parallel variant of the bidiagonal QR iteration with implicit (Wilkinson) shift in a third step [Golub_VanLoan:1996] [Lang:1998] [VanZee_etal:2011] [Reinsch_Richter:2023].

Finally, note that, when OpenMP is used and a large number of processors is available, this routine can be significantly faster than *svd_cmp3()*, *svd_cmp4()* or *svd_cmp5()*, which use the Ralha-Barlow one-sided algorithm in the bidiagonalization step of the SVD [Barlow_etal:2005] [Bosner_Barlow:2007] as the number of required synchronization barriers is small in the bidiagonal reduction step of **svd_cmp()** (e.g., in *bd_cmp()* routine). The situation is reversed if only a small number of processors is available, because the Ralha-Barlow one-sided bidiagonal reduction algorithm is blocked (e.g., "BLAS3").

At the user option, **svd_cmp()** can also compute only the singular values of MAT by the DQDS, bisection or the bidiagonal implicit QR algorithms.

In summary, when OpenMP is used and a large number of processors is available, **svd_cmp()** is one of the best SVD drivers available in STATPACK both in terms of speed and robustness. However, if you are interested only by the leading singular triplets in the SVD of MAT, the *svd_cmp6()*, *svd_cmp7()* and *svd_cmp8()* drivers described below are a better choice in term of speed as subset computations are possible with *svd_cmp6()*, *svd_cmp7()* and *svd_cmp8()*, but not with **svd_cmp()**.

Synopsis:

```
call svd_cmp( mat(:m,:n) , s(:min(m,n)) , failure , v(:n,:min(m,n)) ,␣
↪sort=sort , mul_size=mul_size , maxiter=maxiter , max_francis_steps=max_
↪francis_steps , perfect_shift=perfect_shift , bisect=bisect , dqds=dqds ,␣
↪use_svd2=use_svd2 )

call svd_cmp( mat(:m,:n) , s(:min(m,n)) , failure ,                    ␣
↪sort=sort , mul_size=mul_size , maxiter=maxiter , bisect=bisect , dqds=dqds␣
↪, d=d(:min(m,n)) , e=e(:min(m,n)) , tauq=tauq(:min(m,n)) , taup=taup(:min(m,
↪n)) )
```

Examples:

ex1_svd_cmp.F90

ex2_svd_cmp.F90

ex3_svd_cmp.F90

ex1_bd_deflate2_ter.F90

ex1_random_svd.F90

ex1_random_eig_pos.F90

svd_cmp2()

Purpose:

svd_cmp2() computes the Singular Value Decomposition (SVD) of a real m-by-n matrix MAT. The SVD is written:

$$MAT = U * S * V^T$$

where S is a m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is a m-by-m orthogonal matrix, and V is a n-by-n orthogonal matrix. The diagonal elements of S are the singular values of MAT; they are real and non-negative. The columns of U and V are, respectively, the left and right singular vectors of MAT.

svd_cmp2() computes only the first min(m,n) columns of U and V (e.g., the left and right singular vectors of MAT in the *thin* SVD of MAT). The left singular vectors are returned column-wise and the right singular vectors are returned row-wise in all cases.

This routine uses the same output formats for the SVD factors than the LAPACK SVD routines [Anderson_etal:1999], but is slightly slower than *svd_cmp()*. Otherwise, the same algorithms are used in **svd_cmp2()** and *svd_cmp()* described above.

Finally, note that, when OpenMP is used and a large number of processors is available, this routine can be significantly faster than *svd_cmp3()*, *svd_cmp4()* or *svd_cmp5()*, which use the Ralha-Barlow one-sided algorithm in the bidiagonalization step of the SVD [Barlow_etal:2005] [Bosner_Barlow:2007]. The situation is reversed if only a small number of processors is available.

At the user option, **svd_cmp2()** can also compute only the singular values of MAT by the DQDS, bisection or the bidiagonal implicit QR algorithms.

In summary, when OpenMP is used and a large number of processors is available, **svd_cmp2()** (as *svd_cmp()*) is one of the best SVD drivers available in STATPACK both in terms of speed and robustness. However, if you are interested only by the leading singular triplets in the SVD of MAT, the *svd_cmp6()*, *svd_cmp7()* and *svd_cmp8()* drivers described below are a better choice in term of speed as subset computations are possible with *svd_cmp6()*, *svd_cmp7()* and *svd_cmp8()*, but not with **svd_cmp2()**.

Synopsis:

```
call svd_cmp2( mat(:m,:n) , s(:min(m,n)) , failure , u_vt(:min(m,n),:min(m,
↪n)) , sort=sort , mul_size=mul_size , maxiter=maxiter , max_francis_
↪steps=max_francis_steps , perfect_shift=perfect_shift , bisect=bisect ,␣
↪dqds=dqds , use_svd2=use_svd2 )

call svd_cmp2( mat(:m,:n) , s(:min(m,n)) , failure ,                    ␣
↪       sort=sort , mul_size=mul_size , maxiter=maxiter , bisect=bisect␣
↪, dqds=dqds , d=d(:min(m,n)) , e=e(:min(m,n)) , tauq=tauq(:min(m,n)) ,␣
↪taup=taup(:min(m,n)) )
```

Examples:

ex1_svd_cmp2.F90

ex2_svd_cmp2.F90

svd_cmp3()

Purpose:

svd_cmp3() computes the Singular Value Decomposition (SVD) of a real m-by-n matrix MAT. The SVD is written:

$$MAT = U * S * V^T$$

where S is a m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is a m-by-m orthogonal matrix, and V is a n-by-n orthogonal matrix. The diagonal elements of S are the singular values of MAT; they are real and non-negative. The columns of U and V are, respectively, the left and right singular vectors of MAT.

The routine returns the first min(m,n) singular values and the associated left and right singular vectors corresponding to a *thin* SVD of MAT. The left singular vectors are returned column-wise in all cases and the right singular vectors are returned row-wise if m<n.

The following three-step algorithm is used to compute the *thin* SVD of MAT:

- MAT is first reduced to bidiagonal form B with the help of a blocked (e.g., "BLAS3") and parallel version of the one-sided Ralha-Barlow bidiagonal reduction algorithm [Ralha:2003] [Barlow_etal:2005] [Bosner_Barlow:2007] , with an eventual partial reorthogonalization based on Gram-Schmidt orthogonalization [Stewart:2007]. Furthermore, if m is much larger than n or if n is much larger than m, a preliminary QR or LQ factorization is performed, respectively, and the Ralha-Barlow bidiagonal reduction algorithm is applied to the resulting triangular factor.
- In place accumulation of the left and right orthogonal transformations, used in the reduction of MAT to bidiagonal form B, is performed by fast blocked and parallel algorithms in a second step [Lawson_Hanson:1974] [Walker:1988] [Golub_VanLoan:1996].
- The singular values and vectors of the bidiagonal matrix MAT are computed by a blocked, cache efficient and parallel variant of the bidiagonal QR iteration with implicit (Wilkinson) shift in a third step [Golub_VanLoan:1996] [Lang:1998] [VanZee_etal:2011] [Reinsch_Richter:2023].

Note that for matrices with a very large condition number, **svd_cmp3()** may compute left (right if m<n) singular vectors which are not numerically orthogonal as these singular vectors are computed by a recurrence relationship [Barlow_etal:2005]. A reothogonalization procedure has been implemented in **svd_cmp3()** (e.g. in *bd_cmp2()*) to correct partially this deficiency, but it is not always sufficient to obtain numerically orthogonal left (right if m<n) singular vectors, especially for matrices with a slow decay of singular values near zero. However, this loss of orthogonality concerns only the left (right if m<n) singular vectors associated with the smallest singular values of MAT [Barlow_etal:2005]. The largest singular vectors of MAT are always numerically orthogonal even if MAT is singular or nearly singular.

At the user option, **svd_cmp3()** can also compute only the singular values of MAT by the DQDS, bisection or the bidiagonal implicit QR algorithms.

Finally, note that, when OpenMP is used and only a small number of processors is available, this routine can be significantly faster than *svd_cmp()* or *svd_cmp2()* because of the use of the Ralha-Barlow one-sided algorithm in the bidiagonalization step of the SVD [Barlow_etal:2005] [Bosner_Barlow:2007]. The situation is reversed if a large number of processors is available.

Synopsis:

```
call svd_cmp3( mat(:m,:n) , s(:min(m,n)) , failure , u_v(:min(m,n),:min(m,
↪n)) , sort=sort , maxiter=maxiter , max_francis_steps=max_francis_steps ,␣
↪perfect_shift=perfect_shift , bisect=bisect , dqds=dqds , reortho=reortho ,␣
↪failure_bd=failure_bd )

call svd_cmp3( mat(:m,:n) , s(:min(m,n)) , failure ,                          ␣
↪   sort=sort , maxiter=maxiter , bisect=bisect , dqds=dqds , save_mat=save_
↪mat , reortho=reortho , failure_bd=failure_bd )
```

Examples:

ex1_svd_cmp3.F90

ex2_svd_cmp3.F90

svd_cmp4 ()

Purpose:

svd_cmp4() computes the Singular Value Decomposition (SVD) of a real m-by-n matrix MAT with m>=n. The SVD is written:

$$MAT = U * S * V^T$$

where S is a m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is a m-by-m orthogonal matrix, and V is a n-by-n orthogonal matrix. The diagonal elements of S are the singular values of MAT; they are real and non-negative. The columns of U and V are, respectively, the left and right singular vectors of MAT.

The routine returns the first n singular values and associated left and right singular vectors corresponding to a *thin* SVD of MAT. The left and right singular vectors are returned column-wise.

Optionally, if the logical argument *SING_VEC* is used with the value false, the routine computes only the singular values (by the accurate DQDS algorithm) and the orthogonal matrices Q and P used to reduce MAT to bidiagonal form B. This is useful for computing a partial SVD of the matrix MAT with subroutines *bd_inviter2()* or *bd_deflate2()*, for example, in a second step.

On the other hand, if the logical argument *SING_VEC* is not used or used with the value true, the following four-step algorithm is used to compute the *thin* SVD of MAT:

- MAT is first reduced to bidiagonal form B with the help of a blocked (e.g., "BLAS3") and parallel version of the Ralha-Barlow one-sided bidiagonalization algorithm without any reorthogonalization [Barlow_etal:2005] [Bosner_Barlow:2007]. Furthermore, if m is much larger than n, a preliminary QR factorization is performed and the fast Ralha-Barlow bidiagonal reduction algorithm is applied to the resulting triangular factor.
- In place accumulation of the right orthogonal transformations, used in the reduction of MAT to bidiagonal form B, is performed by a fast blocked and parallel back-transformation algorithm in a second step [Lawson_Hanson:1974] [Walker:1988] [Golub_VanLoan:1996].
- The singular values and right singular vectors of B (which are also those of MAT) are then computed by a blocked, cache efficient and parallel variant of the implicit bidiagonal QR algorithm in a third step [Lawson_Hanson:1974] [Lang:1998] [Golub_VanLoan:1996] [VanZee_etal:2011] .
- The left singular vectors of MAT are finally computed by a matrix multiplication and an orthogonalization step performed with the help of a fast blocked and parallel QR factorization in order to correct for the possible deficiency of the Ralha-Barlow one-sided bidiagonalization algorithm used in the first step if the condition number of MAT is very large.

At the user option, **svd_cmp4()** can also compute only the singular values of MAT by the DQDS, bisection or the bidiagonal implicit QR algorithms.

In serial mode (e.g., without OpenMP), this routine is usually significantly faster than *svd_cmp()* or *svd_cmp2()* for computing the *thin* SVD of MAT because of the use of the Ralha-Barlow one-sided bidiagonalization algorithm without reorthogonalization in the first step [Barlow_etal:2005] [Bosner_Barlow:2007].

Note also that the numerical orthogonality of the left singular vectors computed by **svd_cmp4()** is not affected by the magnitude of the condition number of MAT as in *svd_cmp3()*. In *svd_cmp4()*, this deficiency is fully corrected with the help of the (very fast) final recomputation and orthogonalization step of the left singular vectors (see above), which does not degrade significantly the speed of the subroutine compared to *svd_cmp3()* and also delivers more accurate results than those obtained from this subroutine.

Finally, note that, when OpenMP is used and only a small number of processors is available, this routine can be also significantly faster than *svd_cmp()* or *svd_cmp2()* because of the use of the Ralha-Barlow one-sided algorithm in the bidiagonalization step of the SVD [Barlow_etal:2005] [Bosner_Barlow:2007]. However, the situation is reversed if a large number of processors is available because the number of required synchronization barriers is smaller in the bidiagonal reduction algorithm used in *svd_cmp()* or *svd_cmp2()* than in the Ralha-Barlow one-sided algorithm used in **svd_cmp4()**.

Synopsis:

```
call svd_cmp4( mat(:m,:n) , s(:n) , failure , v(:n,:n) , sort=sort ,
↪ maxiter=maxiter , max_francis_steps=max_francis_steps , perfect_
↪shift=perfect_shift , bisect=bisect , dqds=dqds , sing_vec=sing_vec , gen_
↪p=gen_p , reortho=reortho , failure_bd=failure_bd , d(:n), e(:n) )

call svd_cmp4( mat(:m,:n) , s(:n) , failure ,            sort=sort ,␣
↪maxiter=maxiter , bisect=bisect , dqds=dqds , save_mat=save_mat ,␣
↪reortho=reortho , failure_bd=failure_bd )
```

Examples:

ex1_svd_cmp4.F90

svd_cmp5()

Purpose:

svd_cmp5() computes the Singular Value Decomposition (SVD) of a real m-by-n matrix MAT. The SVD is written:

$$MAT = U * S * V^T$$

where S is a m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is a m-by-m orthogonal matrix, and V is a n-by-n orthogonal matrix. The diagonal elements of S are the singular values of MAT; they are real and non-negative. The columns of U and V are, respectively, the left and right singular vectors of MAT.

This routine returns the first min(m,n) singular values and the associated left and right singular vectors corresponding to a *thin* SVD of MAT. The left and right singular vectors are returned column-wise in all cases.

The following four-step algorithm is used to compute the *thin* SVD of MAT:

- MAT (or its transpose if m<n) is first reduced to bidiagonal form B with the help of a blocked (e.g., "BLAS3") and parallel version of the Ralha-Barlow one-sided bidiagonalization algorithm without reorthogonalization [Barlow_etal:2005] [Bosner_Barlow:2007]. Furthermore, if m is much larger than n or if n is much larger than m, a preliminary QR or LQ factorization is performed, respectively, and the Ralha-Barlow bidiagonal reduction algorithm is applied to the resulting triangular factor.
- In place accumulation of the right orthogonal transformations used in the reduction of MAT (or its transpose if m<n) to bidiagonal form B is performed in a second step [Lawson_Hanson:1974] [Walker:1988] [Golub_VanLoan:1996].
- The singular values and right singular vectors of B (which are also those of MAT) are then computed by a blocked, cache efficient and parallel variant of the implicit bidiagonal QR algorithm in a third step [Lawson_Hanson:1974] [Lang:1998] [Golub_VanLoan:1996] [VanZee_etal:2011] .
- The left (right if m<n) singular vectors of MAT are finally computed by a matrix multiplication and an orthogonalization step performed with the help of a fast blocked and parallel QR factorization in order to correct for the possible deficiency of the Ralha-Barlow one-sided bidiagonalization algorithm used in the first step if the condition number of MAT is very large.

In serial mode (e.g., without OpenMP), this routine is significantly faster than *svd_cmp()* or *svd_cmp2()* because of the use of the Ralha-Barlow one-sided bidiagonalization algorithm without reorthogonalization in the first step of the SVD [Barlow_etal:2005] [Bosner_Barlow:2007]. It is also as fast (or even faster for rank-deficient matrices because reorthogonalization is not performed in the first step above) and more accurate than *svd_cmp3()*, which also uses the Ralha-Barlow one-sided bidiagonalization algorithm.

Furthermore, **svd_cmp5**(), as *svd_cmp4()*, always computes numerical orthogonal singular vectors thanks to the original modifications of the Ralha-Barlow one-sided bidiagonalization algorithm described above. Finally, in contrast to *svd_cmp4()* which uses a similar four-step algorithm, both m>=n and m<n are permitted in **svd_cmp5**().

Finally, note that, when OpenMP is used and only a small number of processors is available, this routine can be also significantly faster than *svd_cmp()* or *svd_cmp2()* because of the use of the Ralha-Barlow one-sided algorithm in the bidiagonalization step of the SVD [Barlow_etal:2005] [Bosner_Barlow:2007]. However, the situation is reversed if a large number of processors is available because the number of required synchronization barriers is smaller in the bidiagonal reduction algorithm used in *svd_cmp()* or *svd_cmp2()* than in the Ralha-Barlow one-sided algorithm used in **svd_cmp5**().

At the user option, **svd_cmp5**() can also compute only the singular values of MAT by the DQDS, bisection or the bidiagonal implicit QR algorithms.

In summary, **svd_cmp5**() is one of the best deterministic SVD drivers available in STATPACK for computing the *thin* SVD of a matrix, both in terms of speed and accuracy, in serial mode or if only a small number of processors is

available in parallel mode (e.g., when OpenMP is used). However, if you are interested only by the leading singular triplets in the SVD of MAT, the *svd_cmp6()*, *svd_cmp7()* and *svd_cmp8()* drivers described below are a better choice in term of speed as subset computations are possible with *svd_cmp6()*, *svd_cmp7()* and *svd_cmp8()*, but not with **svd_cmp5()**.

Synopsis:

```
call svd_cmp5( mat(:m,:n) , s(:min(m,n)) , failure , v(:n,:min(m,n)) ,␣
↪sort=sort , maxiter=maxiter , max_francis_steps=max_francis_steps , perfect_
↪shift=perfect_shift , bisect=bisect , dqds=dqds , use_qr=use_qr , failure_
↪bd=failure_bd )

call svd_cmp5( mat(:m,:n) , s(:min(m,n)) , failure ,                    ␣
↪sort=sort , maxiter=maxiter , bisect=bisect , dqds=dqds , save_mat=save_
↪mat , reortho=reortho , failure_bd=failure_bd )
```

Examples:

ex1_svd_cmp5.F90

svd_cmp6()

Purpose:

svd_cmp6() computes a full or partial Singular Value Decomposition (SVD) of a real m-by-n matrix MAT. The full SVD is written:

$$MAT = U * S * V^T$$

where S is a m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is a m-by-m orthogonal matrix, and V is a n-by-n orthogonal matrix. The diagonal elements of S are the singular values of MAT; they are real and non-negative. The columns of U and V are, respectively, the left and right singular vectors of MAT.

This routine returns the first min(m,n) singular values and the associated left and right singular vectors corresponding to a *thin* SVD of MAT or, alternatively, a truncated SVD of rank nsvd if the optional integer parameter *NSVD* is used in the call to **svd_cmp6()**. The left and right singular vectors are returned column-wise in all cases.

The following four-step algorithm is used to compute the *thin* or *truncated* SVD of MAT:

- MAT (or its transpose if m<n) is first reduced to bidiagonal form B with the help of the fast Ralha-Barlow one-sided bidiagonalization algorithm without reorthogonalization in a first step [Barlow_etal:2005] [Bosner_Barlow:2007].
- The singular values and right singular vectors of B are then computed by the bisection and inverse iteration methods applied to B and the tridiagonal matrix $B^T * B$, respectively, in a second step, see [Golub_VanLoan:1996] for details.
- The right (left if m<n) singular vectors of MAT are then computed by a back-transformation algorithm from those of B in a third step, see [Lawson_Hanson:1974] [Golub_VanLoan:1996] for details.
- Finally, the left (right if m<n) singular vectors of MAT are computed by a matrix multiplication and an orthogonalization step performed with the help of a fast blocked and parallel QR factorization in order to correct for the possible deficiency of the Ralha-Barlow one-sided bidiagonalization algorithm used in the first step if the condition number of MAT is very large.

In serial mode (e.g., without OpenMP), this routine is significantly faster than *svd_cmp()* or *svd_cmp2()* because of the use of the Ralha-Barlow one-sided bidiagonalization algorithm without reorthogonalization in the first step [Barlow_etal:2005] [Bosner_Barlow:2007] and inverse iteration in the second step [Golub_VanLoan:1996]. It is also as fast (or even faster for rank-deficient matrices because reorthogonalization is not performed in the first step above) and more accurate than *svd_cmp3()*, which also uses the Ralha-Barlow one-sided bidiagonalization algorithm. Furthermore, **svd_cmp6()** always computes numerically orthogonal singular vectors (if inverse iteration in the

second step succeeds) thanks to the original modifications of the Ralha-Barlow one-sided bidiagonalization algorithm described above.

In summary, **svd_cmp6()** is one of the good deterministic SVD drivers available in STATPACK for computing the truncated SVD of a matrix both in terms of speed and accuracy.

However, if the *thin* SVD is wanted, *svd_cmp4()* and *svd_cmp5()* drivers described above are better choice in terms of accuracy as these routines are using exactly the same algorithms in the first and last steps of the SVD, but bidiagonal implicit QR iterations for computing the right (left if m<n) singular vectors in the intermediate step of the SVD, which are more accurate than using inverse iterations on the tridiagonal matrix $B^T * B$ as used here in **svd_cmp6()**.

Synopsis:

```
call svd_cmp6( mat(:m,:n) , s(:) , v(:) , failure , sort=sort , nsvd=nsvd␣
↪, maxiter=maxiter , ortho=ortho , backward_sweep=backward_sweep ,␣
↪scaling=scaling , initvec=initvec , use_qr=use_qr , failure_bd=failure_bd ,␣
↪failure_bisect=failure_bisect )
```

Examples:

ex1_svd_cmp6.F90

svd_cmp7()

Purpose:

svd_cmp7() computes a full or partial Singular Value Decomposition (SVD) of a real m-by-n matrix MAT. The full SVD is written:

$$MAT = U * S * V^T$$

where S is a m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is a m-by-m orthogonal matrix, and V is a n-by-n orthogonal matrix. The diagonal elements of S are the singular values of MAT; they are real and non-negative. The columns of U and V are, respectively, the left and right singular vectors of MAT.

This routine returns the min(m,n) singular values and the associated left and right singular vectors corresponding to a *thin* SVD of MAT or, alternatively, the min(m,n) singular values and only the leading singular vectors associated with a truncated SVD of rank nsvd at the user option, where nsvd is the number of columns of the output matrix arguments *U* and *V* used in the call of **svd_cmp7()**. The left and right singular vectors, U and V, are returned column-wise in all cases.

The following four-step algorithm is used to compute the *thin* or *truncated* SVD of MAT:

- MAT is first reduced to bidiagonal form B with the help of a fast, parallel and cache efficient variant of the Golub-Kahan bidiagonal reduction algorithm in a first step [Howell_etal:2008].
- The singular values of the bidiagonal matrix B are computed accurately and efficiently with the Differential Quotient Difference with Shifts (DQDS) algorithm in a second step [Parlett_Marques:2000] [Nakatsukasa_etal:2012] [Li_etal:2014].
- The singular vectors of B are computed accurately by the inverse iteration method applied to the Tridiagonal Golub-Kahan (TGK) form of the bidiagonal matrix B in a third step [Godunov_etal:1993] [Ipsen:1997] [Dhillon:1998] [Marques_Vasconcelos:2017] [Marques_etal:2020].
- The singular vectors of MAT are then computed by fast blocked and parallel back-transformation algorithms from those of B in a final step, see [Walker:1988] [Golub_VanLoan:1996] for more details.

In many cases, when OpenMP activated, and especially when the singular values are relatively well-separated and a large set of processors is available, this routine is significantly faster than other SVD drivers currently available in STATPACK because of the use of a cache efficient and parallel bidiagonal reduction algorithm in the first step, inverse iteration in the third step and highly efficient blocked and parallel back-transformation algorithms for computing the singular vectors of MAT from those of B.

In summary, **svd_cmp7()** is one of the best deterministic SVD drivers available in STATPACK for computing the full or truncated SVD of a matrix in term of speed when OpenMP is used and a large number of processors is available. In its default setting, it is slightly less robust and accurate for computing the *thin* SVD than *svd_cmp()*, *svd_cmp2()* or *svd_cmp5()* drivers described above if the singular values of MAT are not well-separated. However, this drawback can be easily corrected by using the optional logical argument *ORTHO* with the value true in the call to **svd_cmp7()** at the expense of a modest increase in computation time. The increase in computation time is relatively small because the additional full orthogonalization step, which is used when the optional logical argument *ORTHO* is used with the value true, is done with fast blocked and parallel QR algorithms.

Synopsis:

```
call svd_cmp7( mat(:m,:n) , s(:min(m,n)), u(:m,nsvd), v(:n,:nsvd), failure ,␣
↪sort=sort , maxiter=maxiter , ortho=ortho , backward_sweep=backward_sweep ,␣
↪scaling=scaling , initvec=initvec , failure_dqds=failure_dqds )
```

Examples:

ex1_svd_cmp7.F90

svd_cmp8()

Purpose:

svd_cmp8() computes a full or partial Singular Value Decomposition (SVD) of a real m-by-n matrix MAT with m>=:data:*n*. The full SVD is written:

$$MAT = U * S * V^T$$

where S is a m-by-n matrix which is zero except for its n diagonal elements, U is a m-by-m orthogonal matrix, and V is a n-by-n orthogonal matrix. The diagonal elements of S are the singular values of MAT; they are real and non-negative. The columns of U and V are, respectively, the left and right singular vectors of MAT.

This routine returns the n singular values and the associated left and right singular vectors corresponding to a *thin* SVD of MAT or, alternatively, the n singular values and only the leading singular vectors associated with a truncated SVD of rank nsvd at the user option, where nsvd is the number of columns of the output matrix arguments *U* and *V* used in the call of **svd_cmp8()**. The left and right singular vectors are returned column-wise in all cases.

The following four-step algorithm is used to compute the *thin* or *truncated* SVD of MAT:

- MAT is first reduced to bidiagonal form B with the help of a blocked (e.g., "BLAS3") and parallel version of the Ralha-Barlow one-sided bidiagonalization algorithm [Barlow_etal:2005] [Bosner_Barlow:2007].
- The singular values of the bidiagonal matrix B are computed accurately and efficiently with the Differential Quotient Difference with Shifts (DQDS) algorithm in a second step [Parlett_Marques:2000] [Nakatsukasa_etal:2012] [Li_etal:2014].
- The singular vectors of B are computed accurately by the inverse iteration method applied to the Tridiagonal Golub-Kahan (TGK) form of a bidiagonal matrix B in a third step [Godunov_etal:1993] [Ipsen:1997] [Dhillon:1998] [Marques_Vasconcelos:2017] [Marques_etal:2020].
- The singular vectors of MAT are then computed by fast blocked and parallel back-transformation algorithms from those of B in a final step [Golub_VanLoan:1996].

In many cases, when OpenMP activated, and especially when the singular values are relatively well-separated and only a small set of processors is available, this routine is significantly faster than other SVD drivers currently available in STATPACK because of the use of a one-sided bidiagonal reduction algorithm in the first step, inverse iteration in the third step and highly efficient blocked back-transformation algorithms for computing the singular vectors of MAT from those of B.

In summary, **svd_cmp8()** is one of the best deterministic SVD drivers available in STATPACK for computing the full or truncated SVD of a matrix in term of speed when OpenMP is used and only a small number of processors is available. In its default setting, it is slightly less robust and accurate for computing the *thin* SVD than *svd_cmp()*,

svd_cmp2() or *svd_cmp5()* drivers described above if the singular values of MAT are not well-separated. However, this drawback can be easily corrected by using the optional logical argument *ORTHO* with the value true in the call to **svd_cmp8()** at the expense of a modest increase in computation time. The increase in computation time is relatively small because the additional full orthogonalization step, which is used when the optional logical argument *ORTHO* is used with the value true, is done with fast blocked and parallel QR algorithms.

Synopsis:

```
call svd_cmp8( mat(:m,:n) , s(:min(m,n)), u(:m,nsvd), v(:n,:nsvd), failure ,␣
↪sort=sort , maxiter=maxiter , ortho=ortho , backward_sweep=backward_sweep ,␣
↪scaling=scaling , initvec=initvec , reortho=reortho, failure_bd=failure_bd ,
↪ failure_dqds=failure_dqds )
```

Examples:

ex1_svd_cmp8.F90

rqr_svd_cmp()

Purpose:

rqr_svd_cmp() computes approximations of the nsvd largest singular values and associated left and right singular vectors of a full real m-by-n matrix MAT using a three-step procedure, which can be termed a QR-SVD algorithm [Xiao_etal:2017]:

- first, a randomized (or deterministic) partial QR factorization with Column Pivoting (QRCP) of MAT is computed;
- in a second step, a Singular Value Decomposition (SVD) of the (permuted) upper triangular or trapezoidal (e.g., if m<n) factor R in this QR decomposition is computed. The singular values and right singular vectors of this SVD of R are also estimates of the singular values and right singular vectors of MAT;
- estimates of the associated left singular vectors of MAT are then obtained by pre-multiplying the left singular of R by the orthogonal matrix Q in the initial QR decomposition.

The routine returns these approximations of the first nsvd singular values and the associated left and right singular vectors corresponding to a partial SVD of MAT. The singular vectors are returned column-wise in all cases.

This routine is always significantly faster than the *svd_cmp()*, *svd_cmp2()*, *svd_cmp3()*, *svd_cmp4()*, *svd_cmp5()*, *svd_cmp6()*, *svd_cmp7()*, *svd_cmp8()* *standard* SVD drivers or the *bd_inviter2()* inverse iteration and *bd_deflate2()* deflation drivers because of the use of a cheap QRCP in the first step of the algorithm [Xiao_etal:2017]. However, the computed nsvd largest singular values and associated left and right singular vectors are only approximations of the *true* largest singular values and vectors. The accuracy is also less than in the randomized *rsvd_cmp()* and *rqlp_svd_cmp()* subroutines described below, but **rqr_svd_cmp()** is significantly faster than *rsvd_cmp()* and *rqlp_svd_cmp()* and is thus well adapted if you need a fast, but rough, estimate of a truncated SVD of MAT.

Synopsis:

```
call rqr_svd_cmp( mat(:m,:n) , s(:nsvd) , failure , v(:n,:nsvd) , random_
↪qr=random_qr , truncated_qr=truncated_qr , rng_alg=rng_alg , blk_size=blk_
↪size , nover=nover , nover_svd=nover_svd , maxiter=maxiter , max_francis_
↪steps=max_francis_steps , perfect_shift=perfect_shift , bisect=bisect ,␣
↪dqds=dqds )
```

Examples:

ex1_rqr_svd_cmp.F90

rsvd_cmp()

Purpose:

rsvd_cmp() computes approximations of the nsvd largest singular values and associated left and right singular vectors of a full real m-by-n matrix MAT by *(i)* using randomized power, subspace or block Krylov iterations in order to compute an orthonormal matrix whose range approximates the range of MAT, *(ii)* projecting MAT onto this orthonormal basis and, finally, *(iii)* computing the *standard* SVD of this matrix projection to estimate a truncated SVD of MAT [Halko_etal:2011] [Musco_Musco:2015] [Martinsson:2019].

This routine is always significantly faster than the *svd_cmp()*, *svd_cmp2()*, *svd_cmp3()*, *svd_cmp4()*, *svd_cmp5()*, *svd_cmp6()*, *svd_cmp7()*, *svd_cmp8()* *standard* SVD drivers or the *bd_inviter2()* inverse iteration and *bd_deflate2()* deflation drivers because of the use of very fast randomized algorithms in the first step [Halko_etal:2011] [Gu:2015] [Musco_Musco:2015] [Martinsson:2019]. However, the computed nsvd largest singular values and associated left and right singular vectors are only approximations of the *true* largest singular values and vectors.

The routine returns approximations to the first nsvd singular values and the associated left and right singular vectors corresponding to a partial SVD of MAT. The singular vectors are returned column-wise in all cases. The accuracy is greater than in the randomized *rqr_svd_cmp()* subroutine described above, but **rsvd_cmp()** is slower than *rqr_svd_cmp()*.

Synopsis:

```
call rsvd_cmp( mat(:m,:n) , s(:nsvd) , leftvec(:m,:nsvd) , rightvec(:n,
↪:nsvd) , failure=failure , niter=niter , nover=nover , ortho=ortho , extd_
↪samp=extd_samp , rng_alg=rng_alg , maxiter=maxiter , max_francis_steps=max_
↪francis_steps , perfect_shift=perfect_shift , bisect=bisect , dqds=dqds )
```

Examples:

ex1_rsvd_cmp.F90

rqlp_svd_cmp()

Purpose:

rqlp_svd_cmp() computes approximations of the nsvd largest singular values and associated left and right singular vectors of a full real m-by-n matrix MAT using a four-step procedure, which can be termed a QLP-SVD algorithm [Duersch_Gu:2017] [Feng_etal:2019] [Duersch_Gu:2020]:

- first, a randomized (or deterministic) partial QLP factorization of MAT is computed;
- in a second step, the matrix product MAT * P^T is computed and a number of QR-QL iterations are performed on it to improve the estimates of the principal row and columns subspaces of MAT;
- in a third step, a Singular Value Decomposition (SVD) of this matrix product MAT * P^T is computed. The singular values and left singular vectors in this SVD are also estimates of the singular values and left singular vectors of MAT;
- in a final step, estimates of the associated right singular vectors of MAT are then obtained by pre-multiplying P^T by the right singular vectors in the SVD of this matrix product.

The routine returns accurate approximations to the first nsvd singular values and the associated left and right singular vectors corresponding to a partial SVD of MAT. The singular vectors are returned column-wise in all cases.

This routine is always significantly faster than the *svd_cmp()*, *svd_cmp2()*, *svd_cmp3()*, *svd_cmp4()*, *svd_cmp5()*, *svd_cmp6()*, *svd_cmp7()*, *svd_cmp8()* *standard* SVD drivers or the *bd_inviter2()* inverse iteration and *bd_deflate2()* deflation drivers because of the use of a cheap QLP in the first step of the algorithm [Duersch_Gu:2017] [Feng_etal:2019] [Duersch_Gu:2020]. However, the computed nsvd largest singular values and associated left and right singular vectors are only (very good) approximations of the *true* largest singular values and vectors. The accuracy is always greater than in the randomized *rqr_svd_cmp()* and *rsvd_cmp()* subroutines described above, but **rqlp_svd_cmp()** is usually slower than *rsvd_cmp()* and *rqr_svd_cmp()*, and is more memory demanding.

Synopsis:

```
call rqlp_svd_cmp( mat(:m,:n) , s(:nsvd) , leftvec(:m,:nsvd) , rightvec(:n,
↪:nsvd) , failure , niter=niter , random_qr=random_qr , truncated_
↪qr=truncated_qr , rng_alg=rng_alg , blk_size=blk_size , nover=nover , nover_
↪svd=nover_svd , maxiter=maxiter , max_francis_steps=max_francis_steps ,␣
↪perfect_shift=perfect_shift , bisect=bisect , dqds=dqds )
```

Examples:

ex1_rqlp_svd_cmp.F90

rqlp_svd_cmp2()

Purpose:

rqlp_svd_cmp2() computes approximations of the nsvd largest singular values and associated left and right singular vectors of a full real m-by-n matrix MAT using a four-step procedure, which can be termed a QLP-SVD algorithm [Mary_etal:2015] [Duersch_Gu:2017] [Feng_etal:2019] [Duersch_Gu:2020]:

- first, a very fast approximate and randomized partial QLP factorization of MAT is computed using results in [Mary_etal:2015];
- in a second step, the matrix product MAT * P^T is computed and a number of QR-QL iterations are performed on it to improve the estimates of the principal row and columns subspaces of MAT;
- in a third step, a Singular Value Decomposition (SVD) of this matrix product MAT * P^T is computed. The singular values and left singular vectors in this SVD are also estimates of the singular values and left singular vectors of MAT;
- in a final step, estimates of the associated right singular vectors of MAT are then obtained by pre-multiplying P^T by the right singular vectors in the SVD of this matrix product.

The routine returns accurate approximations to the first nsvd singular values and the associated left and right singular vectors corresponding to a partial SVD of MAT. The singular vectors are returned column-wise in all cases.

This routine is always significantly faster than the *svd_cmp()*, *svd_cmp2()*, *svd_cmp3()*, *svd_cmp4()*, *svd_cmp5()*, *svd_cmp6()*, *svd_cmp7()*, *svd_cmp8()* *standard* SVD drivers or the *bd_inviter2()* inverse iteration and *bd_deflate2()* deflation drivers because of the use of a very cheap approximate QLP in the first step of the algorithm [Mary_etal:2015] [Duersch_Gu:2017] [Feng_etal:2019]. However, the computed nsvd largest singular values and associated left and right singular vectors are only (very good) approximations of the *true* largest singular values and vectors. The accuracy is always greater than in the randomized *rqr_svd_cmp()* and *rsvd_cmp()* subroutines described above and **rqlp_svd_cmp2**() can be as fast than *rsvd_cmp()*.

Thanks to the very fast approximate and randomized partial QLP used in the first step, **rqlp_svd_cmp2**() is also significantly faster than *rqlp_svd_cmp()* described above and is also less memory demanding, without degrading too much the accuracy of the results. Thus, **rqlp_svd_cmp2**() is one of the best randomized SVD drivers available in STATPACK.

Synopsis:

```
call rqlp_svd_cmp2( mat(:m,:n) , s(:nsvd) , leftvec(:m,:nsvd) , rightvec(:n,
↪:nsvd) , failure , niter=niter , rng_alg=rng_alg , nover=nover , nover_
↪svd=nover_svd , maxiter=maxiter , max_francis_steps=max_francis_steps ,␣
↪perfect_shift=perfect_shift , bisect=bisect , dqds=dqds )
```

Examples:

ex1_rqlp_svd_cmp2.F90

rqr_svd_cmp_fixed_precision()

Purpose:

rqr_svd_cmp_fixed_precision() computes approximations of the nsvd largest singular values and associated left and right singular vectors of a full real m-by-n matrix MAT using a three-step procedure, which can be termed a QR-SVD algorithm [Xiao_etal:2017]:

- first, a randomized (or deterministic) partial QR factorization with Column Pivoting (QRCP) of MAT is computed;
- in a second step, a Singular Value Decomposition (SVD) of the (permuted) upper triangular or trapezoidal (e.g., if m<n) factor R in this QR decomposition is computed. The singular values and right singular vectors of this SVD of R are also estimates of the singular values and right singular vectors of MAT;
- estimates of the associated left singular vectors of MAT are then obtained by pre-multiplying the left singular of R by the orthogonal matrix Q Q in the initial QR decomposition.

nsvd is the target rank of the partial SVD, which is sought, and this partial SVD must have an approximation error which fulfills:

$$||MAT - rSVD||_F <= ||MAT||_F.relerr$$

, where rSVD is the computed partial SVD approximation, $||||_F$ is the Frobenius norm and relerr is a prescribed accuracy tolerance for the relative error of the computed partial SVD approximation in the Frobenius norm, specified as an argument (e.g., argument *RELERR*) in the call to **rqr_svd_cmp_fixed_precision()**.

In other words, nsvd is not known in advance and is determined in the subroutine, which is in contrast to *rqr_svd_cmp()* subroutine in which nsvd is an input argument. This explains why the output real array arguments *S* and *V*, which contain the computed singular values and the associated right singular vectors in the partial SVD on exit, must be declared in the calling program as pointers.

On exit, nsvd is equal to the size of the output real pointer argument *S*, which contains the computed singular values and the relative error in the Frobenius norm of the computed partial SVD approximation is output in argument *RELERR*.

As *rqr_svd_cmp()*, this routine is always significantly faster than the *svd_cmp()*, *svd_cmp2()*, *svd_cmp3()*, *svd_cmp4()*, *svd_cmp5()*, *svd_cmp6()*, *svd_cmp7()*, *svd_cmp8()* *standard* SVD drivers or the *bd_inviter2()* inverse iteration and *bd_deflate2()* deflation drivers because of the use of a cheap QRCP in the first step of the algorithm [Xiao_etal:2017]. However, the computed nsvd largest singular values and associated left and right singular vectors are only approximations of the *true* largest singular values and vectors. The accuracy is less than in the randomized *rsvd_cmp_fixed_precision()* and *rqlp_svd_cmp_fixed_precision()* subroutines described below, but **rqr_svd_cmp_fixed_precision()** is significantly faster than *rsvd_cmp_fixed_precision()* and *rqlp_svd_cmp_fixed_precision()* and is thus well adapted if you need a fast, but rough, estimate of a truncated SVD of MAT.

Note, finally, that if you already know the rank of the partial SVD of MAT you are seeking, it is better to use *rqr_svd_cmp()* rather than **rqr_svd_cmp_fixed_precision()** as *rqr_svd_cmp()* is faster and more accurate.

Synopsis:

```
call rqr_svd_cmp_fixed_precision( mat(:m,:n) , relerr , s(:) , failure ,
↪ v(:,:) , random_qr=random_qr , rng_alg=rng_alg , blk_size=blk_size ,␣
↪nover=nover , maxiter=maxiter , max_francis_steps=max_francis_steps ,␣
↪perfect_shift=perfect_shift , bisect=bisect , dqds=dqds )
```

Examples:

ex1_rqr_svd_cmp_fixed_precision.F90

rsvd_cmp_fixed_precision()

Purpose:

rsvd_cmp_fixed_precision() computes approximations of the top nsvd singular values and associated left and right singular vectors of a full real m-by-n matrix MAT using randomized power or subspace iterations as in *rsvd_cmp()*

described [Halko_etal:2011] [Li_etal:2017] [Yu_etal:2018].

nsvd is the target rank of the partial Singular Value Decomposition (SVD), which is sought, and this partial SVD must have an approximation error which fulfills:

$$||MAT - rSVD||_F <= ||MAT||_F.relerr$$

, where rSVD is the computed partial SVD approximation, $||||_F$ is the Frobenius norm and relerr is a prescribed accuracy tolerance for the relative error of the computed partial SVD approximation in the Frobenius norm, specified as an argument (e.g., argument *RELERR*) in the call to **rsvd_cmp_fixed_precision()**.

In other words, nsvd is not known in advance and is determined in the subroutine, which is in contrast to *rsvd_cmp()* subroutine in which nsvd is an input argument. This explains why the output real array arguments *S*, *LEFTVEC* and *RIGHTVEC*, which contain the computed singular triplets of the partial SVD on exit, must be declared in the calling program as pointers.

rsvd_cmp_fixed_precision() searches incrementally the best (e.g., smallest) partial SVD approximation, which fulfills the prescribed accuracy tolerance for the relative error based on an improved version of the randQB_FP algorithm described in [Yu_etal:2018]. See also [Martinsson_Voronin:2016]. More precisely, the rank of the truncated SVD approximation is increased progressively of *BLK_SIZE* by *BLK_SIZE* until the prescribed accuracy tolerance is satisfied and then improved and adjusted precisely by additional subspace iterations (as specified by the optional *NITER_QB* integer argument) to obtain the smallest partial SVD approximation, which satisfies the prescribed tolerance.

On exit, nsvd is equal to the size of the output real pointer argument *S*, which contains the computed singular values and the relative error in the Frobenius norm of the computed partial SVD approximation is output in argument *RELERR*.

As *rsvd_cmp()*, this routine is always significantly faster than the *svd_cmp()*, *svd_cmp2()*, *svd_cmp3()*, *svd_cmp4()*, *svd_cmp5()*, *svd_cmp6()*, *svd_cmp7()*, *svd_cmp8()* *standard* SVD drivers or the *bd_inviter2()* inverse iteration and *bd_deflate2()* deflation drivers because of the use of very fast randomized algorithms [Halko_etal:2011] [Gu:2015] [Li_etal:2017] [Yu_etal:2018]. However, the computed nsvd largest singular values and associated left and right singular vectors are only approximations of the *true* largest singular values and vectors.

Note, finally, that if you already know the rank of the partial SVD of MAT you are seeking, it is better to use *rsvd_cmp()* rather than **rsvd_cmp_fixed_precision()** as *rsvd_cmp()* is faster and slightly more accurate.

Synopsis:

```
call rsvd_cmp_fixed_precision( mat(:m,:n) , relerr , s(:) , leftvec(:,:)␣
↪, rightvec(:,:) , failure_relerr=failure_relerr , failure=failure ,␣
↪niter=niter , blk_size=blk_size , maxiter_qb=maxiter_qb , ortho=ortho ,␣
↪reortho=reortho , niter_qb=niter_qb , rng_alg=rng_alg , maxiter=maxiter␣
↪, max_francis_steps=max_francis_steps , perfect_shift=perfect_shift ,␣
↪bisect=bisect , dqds=dqds )
```

Examples:

ex1_rsvd_cmp_fixed_precision.F90

rqlp_svd_cmp_fixed_precision()

Purpose:

rqlp_svd_cmp_fixed_precision() computes approximations of the nsvd largest singular values and associated left and right singular vectors of a full real m-by-n matrix MAT using a four-step procedure, which can be termed a QLP-SVD algorithm [Duersch_Gu:2017] [Feng_etal:2019] [Duersch_Gu:2020]:

- first, a randomized (or deterministic) partial QLP factorization of MAT is computed;
- in a second step, the matrix product MAT * P^T is computed and a number of QR-QL iterations are performed on it to improve the estimates of the principal row and columns subspaces of MAT;

- in a third step, a Singular Value Decomposition (SVD) of this matrix product MAT * P^T is computed. The singular values and left singular vectors in this SVD are also estimates of the singular values and left singular vectors of MAT;
- in a final step, estimates of the associated right singular vectors of MAT are then obtained by pre-multiplying P^T by the right singular vectors in the SVD of this matrix product.

nsvd is the target rank of the partial SVD, which is sought, and this partial SVD must have an approximation error which fulfills:

$$||MAT - rSVD||_F <= ||MAT||_F.relerr$$

, where rSVD is the computed partial SVD approximation, $||||_F$ is the Frobenius norm and relerr is a prescribed accuracy tolerance for the relative error of the computed partial SVD approximation in the Frobenius norm, specified as an argument (e.g., argument *RELERR*) in the call to **rqlp_svd_cmp_fixed_precision**().

In other words, nsvd is not known in advance and is determined in the subroutine, which is in contrast to *rqlp_svd_cmp()* subroutine in which nsvd is an input argument. This explains why the output real array arguments *S*, *LEFTVEC* and *RIGHTVEC*, which contain the computed singular triplets in the partial SVD on exit, must be declared in the calling program as pointers.

On exit, nsvd is equal to the size of the output real pointer argument *S*, which contains the computed singular values and the relative error in the Frobenius norm of the computed partial SVD approximation is output in argument *RELERR*.

As *rqlp_svd_cmp()*, this routine is always significantly faster than the *svd_cmp()*, *svd_cmp2()*, *svd_cmp3()*, *svd_cmp4()*, *svd_cmp5()*, *svd_cmp6()*, *svd_cmp7()*, *svd_cmp8()* *standard* SVD drivers or the *bd_inviter2()* inverse iteration and *bd_deflate2()* deflation drivers because of the use of a cheap QLP in the first step of the algorithm [Duersch_Gu:2017] [Feng_etal:2019] [Duersch_Gu:2020]. However, the computed nsvd largest singular values and associated left and right singular vectors are only (very good) approximations of the *true* largest singular values and vectors. The accuracy is always greater than in the randomized *rqr_svd_cmp_fixed_precision()* and *rsvd_cmp_fixed_precision()* subroutines described above, but **rqlp_svd_cmp_fixed_precision**() is usually slower than *rsvd_cmp_fixed_precision()* and *rqr_svd_cmp_fixed_precision()*.

Note, finally, that if you already know the rank of the partial SVD of MAT you are seeking, it is better to use *rqlp_svd_cmp()* or *rqlp_svd_cmp2()* rather than **rqlp_svd_cmp_fixed_precision**() as *rqlp_svd_cmp()* and *rqlp_svd_cmp2()* are faster and more accurate.

Synopsis:

```
call rqlp_svd_cmp_fixed_precision( mat(:m,:n) , relerr , s(:) , leftvec(:,:) ,
↪ rightvec(:,:) , failure , niter=niter , random_qr=random_qr , rng_alg=rng_
↪alg , blk_size=blk_size , nover=nover , maxiter=maxiter , max_francis_
↪steps=max_francis_steps , perfect_shift=perfect_shift , bisect=bisect ,␣
↪dqds=dqds )
```

Examples:

ex1_rqlp_svd_cmp_fixed_precision.F90

reig_pos_cmp()

Purpose:

reig_pos_cmp() computes approximations of the neig largest eigenvalues and associated eigenvectors of a full real n-by-n symmetric positive semi-definite matrix MAT using randomized power, subspace or block Krylov iterations [Halko_etal:2011] [Musco_Musco:2015] [Martinsson:2019] and, at the user option, the Nystrom method [Li_etal:2017], [Martinsson:2019] [Halko_etal:2011]. The Nystrom method provides more accurate results for positive semi-definite matrices [Halko_etal:2011] [Martinsson:2019].

The Nystrom method will be selected in **reig_pos_cmp()** if the optional logical argument *USE_NYSTROM* is used with the value `true` (this is the default), otherwise the standard EigenValue Decomposition (EVD) algorithm will be used in the last step of the randomized algorithm.

This routine is always significantly faster than *eig_cmp()*, *eig_cmp2()*, *eig_cmp3()* or *trid_inviter()* and *trid_deflate()* in module *Eig_Procedures* because of the use of very fast randomized algorithms [Halko_etal:2011] [Gu:2015] [Musco_Musco:2015] [Martinsson:2019]. However, the computed `neig` largest eigenvalues and eigenvectors are only approximations of the *true* largest eigenvalues and eigenvectors.

The routine returns approximations to the first `neig` eigenvalues and the associated eigenvectors corresponding to a partial EVD of a symmetric positive semi-definite matrix `MAT`.

Synopsis:

```
call reig_pos_cmp( mat(:n,:n) , eigval(:neig) , eigvec(:n,:neig) ,␣
↪failure=failure , niter=niter , nover=nover , ortho=ortho , extd_samp=extd_
↪samp , use_nystrom=use_nystrom , rng_alg=rng_alg , maxiter=maxiter ,␣
↪max_francis_steps=max_francis_steps , perfect_shift=perfect_shift ,␣
↪bisect=bisect , dqds=dqds )
```

Examples:

ex1_reig_pos_cmp.F90

qlp_cmp()

Purpose:

qlp_cmp() computes a partial or complete QLP factorization of a `m`-by-`n` matrix `MAT` [Stewart:1999b]:

$$MAT \simeq Q * L * P$$

where `Q` and `P` are, respectively, a `m`-by-`krank` matrix with orthonormal columns and a `krank`-by-`n` matrix with orthonormal rows (and `krank<=min(m,n)`), and `L` is a `krank`-by-`krank` lower triangular matrix. If `krank=min(m,n)`, the QLP factorization is complete.

The QLP factorization can obtained by a two-step algorithm:

- first, a partial (or complete) QR factorization with Column Pivoting (QRCP) of `MAT` is computed;
- in a second step, a LQ decomposition of the (permuted) upper triangular or trapezoidal (e.g., if `n>m`) factor, `R`, of this QR decomposition is computed.

By default, a standard deterministic QRCP is used in the first phase of the QLP algorithm [Golub_VanLoan:1996]. However, if the optional logical argument *RANDOM_QR* is used with the value `true`, an alternate fast randomized (partial or full) QRCP is used in the first phase of the QLP algorithm [Duersch_Gu:2017] [Xiao_etal:2017] [Duersch_Gu:2020].

At the user option, the QLP factorization can also be only partial, e.g., the subroutine stops the computations when the numbers of columns of `Q` and and the rows of `P` is equal to a predefined value equals to `krank = size( BETA ) = size( TAU )`.

By default, **qlp_cmp()** outputs the QLP decomposition in factored form in arguments *MAT*, *BETA*, *TAU* and *LMAT*, but `Q` and `P` can also be generated at the user option.

The QLP decomposition provides a reasonable and cheap estimate of the Singular Value Decomposition (SVD) of a matrix when this matrix has a low rank or a significant gap in its singular values spectrum [Stewart:1999b] [Huckaby_Chan:2003] [Huckaby_Chan:2005].

Synopsis:

```
call qlp_cmp( mat(:m,:n) , beta(:krank) , tau(:krank) , lmat(:krank,:krank)␣
↪, qmat=qmat(:m,:krank) , pmat=pmat(:krank,:n) , random_qr=random_qr␣
↪, truncated_qr=truncated_qr , rng_alg=rng_alg , blk_size=blk_size ,␣
↪nover=nover )
```

Examples:

ex1_qlp_cmp.F90

qlp_cmp2()

Purpose2:

qlp_cmp2() computes an improved partial or complete QLP factorization of a m-by-n matrix MAT [Stewart:1999b]:

$$MAT \simeq Q * L * P$$

where Q and P are, respectively, a m-by-krank matrix with orthonormal columns and a krank-by-n matrix with orthonormal rows (and krank<=min(m,n)), and L is a krank-by-krank lower triangular matrix. If krank=min(m,n), the QLP factorization is complete.

In contrast to *qlp_cmp()*, the QLP factorization in **qlp_cmp2**() is obtained by a three-step algorithm:

- first, a partial (or complete) QR factorization with Column Pivoting (QRCP) of MAT is computed;
- in a second step, a LQ decomposition of the (permuted) upper triangular or trapezoidal (e.g., if n> m) factor, R, of this QR decomposition is computed;
- in a final step, *NITER_QRQL* QR-QL iterations can be performed on the L factor in this LQ decomposition to improve the accuracy of the diagonal elements of L (the so called *L-values*) as estimates of the singular values of MAT [Stewart:1999b] [Huckaby_Chan:2003] [Huckaby_Chan:2005] [Wu_Xiang:2020].

By default, a standard deterministic QRCP is used in the first phase of the QLP algorithm [Golub_VanLoan:1996]. However, if the optional logical argument *RANDOM_QR* is used with the value true, an alternate fast randomized (partial or full) QRCP is used in the first phase of the QLP algorithm [Duersch_Gu:2017] [Xiao_etal:2017] [Duersch_Gu:2020].

At the user option, the QLP factorization can also be only partial, e.g., the subroutine stops the computations when the numbers of columns of Q and of the rows of P is equal to a predefined value equals to krank = size(LMAT,1) = size(LMAT,2).

qlp_cmp2() outputs the QLP decomposition of MAT in standard form in the matrix arguments *LMAT*, *QMAT* and *PMAT*.

The main differences between **qlp_cmp2**() and *qlp_cmp()* described above, are in this explicit output format of the QLP decomposition and the possibility of improving the accuracy of the *L-values* of the initial QLP decomposition by additional QR-QL iterations in **qlp_cmp2**() (as specified by the optional integer parameter *NITER_QRQL* of **qlp_cmp2**()).

The (improved) QLP decomposition provides a reasonable and cheap estimate of the Singular Value Decomposition (SVD) of a matrix when this matrix has a low rank or a significant gap in its singular values spectrum [Stewart:1999b] [Huckaby_Chan:2003] [Huckaby_Chan:2005].

Synopsis:

```
call qlp_cmp2( mat(:m,:n) , lmat(:krank,:krank) , qmat(:m,:krank) ,␣
↪pmat(:krank,:n) , niter_qrql=niter_qrql , random_qr=random_qr , truncated_
↪qr=truncated_qr , rng_alg=rng_alg , blk_size=blk_size , nover=nover )
```

Examples:

ex1_qlp_cmp2.F90

rqlp_cmp()

Purpose2:

rqlp_cmp() computes an approximate randomized partial QLP factorization of a m-by-n matrix MAT [Stewart:1999b] [Wu_Xiang:2020]:

$$MAT \simeq Q * L * P$$

where Q and P are, respectively, a m-by-krank matrix with orthonormal columns and a krank-by-n matrix with orthonormal rows (and krank<=min(m,n)), and L is a krank-by-krank lower triangular matrix.

In contrast to *qlp_cmp()* and *qlp_cmp2()*, the QLP factorization in **rqlp_cmp()** is obtained by a four-step algorithm:

- first, a partial QB factorization of MAT is computed with the help of a randomized algorithm [Martinsson_Voronin:2016] [Wu_Xiang:2020]

  $$MAT \simeq Q * B$$

  where Q is m-by-krank matrix with orthonormal columns and B is a full krank-by-n matrix such that the matrix product Q*B is a good approximation of MAT in the spectral or Frobenius norm;
- in a second step, a partial (or complete) QR factorization with Column Pivoting (QRCP) of B is computed and Q is post-multiplied by the krank-by-krank orthogonal matrix in this QR decomposition of B;
- in a third step, a LQ decomposition of the (permuted) upper trapezoidal factor, R, of this QR decomposition of B is computed;
- in a final step, *NITER_QRQL* QR-QL iterations can be performed on the L factor in this LQ decomposition to improve the accuracy of the diagonal elements of L (the so called *L-values*) as estimates of the singular values of MAT [Stewart:1999b] [Huckaby_Chan:2003] [Huckaby_Chan:2005] [Wu_Xiang:2020].

The QLP factorization in **rqlp_cmp()** is only partial, e.g., the subroutine stops the computations when the numbers of columns of Q and of the rows of P is equal to a predefined value equals to `krank = size(LMAT,1) = size(LMAT,2)`.

rqlp_cmp() outputs the QLP decomposition in standard form in the matrix arguments *LMAT*, *QMAT* and *PMAT*.

The main differences between **rqlp_cmp()** and *qlp_cmp()* and *qlp_cmp2()* described above, are in the use of a preliminary QB decomposition of MAT to identify the principal subspace of the columns of MAT before computing the QLP decomposition (of the projection of MAT onto this subspace). This delivers usually higher accuracy in the final partial QLP decomposition than those obtained in *qlp_cmp()* and *qlp_cmp2()*.

This randomized partial QLP decomposition provides a reasonable and cheap estimate of the Singular Value Decomposition (SVD) of a matrix when this matrix has a low rank or a significant gap in its singular values spectrum [Stewart:1999b] [Huckaby_Chan:2003] [Huckaby_Chan:2005] [Wu_Xiang:2020].

Synopsis:

```
call rqlp_cmp( mat(:m,:n) , lmat(:krank,:krank) , qmat(:m,:krank) ,␣
↪pmat(:krank,:n) , niter=niter , rng_alg=rng_alg , ortho=ortho , niter_
↪qrql=niter_qrql )
```

Examples:

ex1_rqlp_cmp.F90

singvalues()

Purpose:

singvalues() computes the singular values of a real m-by-n matrix MAT. The Singular Value decomposition (SVD) is written

$$MAT = U * S * V^T$$

where S is a m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is a m-by-m orthogonal matrix, and V is a n-by-n orthogonal matrix. The diagonal elements of S are the singular values of MAT; they are real and non-negative.

The inpit matrix MAT is first reduced to bidiagonal B form by a one- or two-step algorithm depending on its shape and the singular values of B (which are also those of MAT) are computed by the Differential Quotient Difference with Shifts (DQDS) algorithm [Fernando_Parlett:1994] [Li_etal:2014] or the bidiagonal QR algorithm [Lawson_Hanson:1974] [Golub_VanLoan:1996] [Reinsch_Richter:2023] at the user option. The default is to use the DQDS algorithm as it is usually faster and more accurate than the QR algorithm.

Synopsis:

```
singval(:min(m,n)) = singvalues( mat(:m,:n) , sort=sort , mul_size=mul_size ,␣
↪maxiter=maxiter , dqds=dqds )
```

Examples:

ex1_singvalues.F90

select_singval_cmp()

Purpose:

select_singval_cmp() computes all or some of the greatest singular values of a real m-by-n matrix MAT.

The Singular Value decomposition (SVD) is written:

$$MAT = U * S * V^T$$

where S is a m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is a m-by-m orthogonal matrix, and V is a n-by-n orthogonal matrix. The diagonal elements of S are the singular values of MAT; they are real and non-negative.

Both one-step and a two-step algorithms are available in **select_singval_cmp()** for the preliminary reduction of the input matrix MAT to bidiagonal form.

In the one-step algorithm, the original matrix MAT is directly reduced to upper or lower bidiagonal form BD by an orthogonal transformation:

$$Q^T * MAT * P = BD$$

where Q and P are orthogonal (see [Golub_VanLoan:1996] [Lawson_Hanson:1974] [Howell_etal:2008]).

In the two-step algorithm, the original matrix MAT is also reduced to upper bidiagonal form BD, but if:

- m >= n, a QR factorization of the real m-by-n matrix MAT is first computed

  $$MAT = O * R$$

  where O is orthogonal and R is upper triangular. In a second step, the n-by-n upper triangular matrix R is reduced to upper bidiagonal form BD by an orthogonal transformation:

  $$Q^T * R * P = BD$$

  where Q and P are orthogonal and BD is an upper bidiagonal matrix.

- m < n, an LQ factorization of the real m-by-n matrix MAT is first computed

  $$MAT = L * O$$

  where O is orthogonal and L is lower triangular. In a second step, the m-by-m lower triangular matrix L is also reduced to upper bidiagonal form BD by an orthogonal transformation :

  $$Q^T * L * P = BD$$

  where Q and P are orthogonal and BD is also an upper bidiagonal matrix.

This two-step reduction algorithm will be more efficient if m is much larger than n or if n is much larger than m.

In both the one-step and two-step algorithms, the singular values S of the bidiagonal matrix BD, which are also the singular values of MAT, are then computed by the DQDS algorithm [Parlett_Marques:2000] [Li_etal:2014] or a bisection algorithm applied to the Tridiagonal Golub-Kahan (TGK) form of the bidiagonal matrix BD (see [Fernando:1998]; Sec.3.3).

Both the DQDS and bisection algorithms are able to compute the singular values of BD to high relative accuracy, bisection is more accurate, but can be much slower than DQDS when all singular values are computed for some bidiagonal matrices. Thus, the default is to use the DQDS algorithm if all the singular values are requested and bisection if only selected singular values are requested, but the user can select the desired algorithm with the optional logical argument *DQDS* in the call to **select_singval_cmp()**. Note that the DQDS algorithm always computes all the singular values of BD, subset computation of the singular values is only possible with the bisection algorithm.

The routine outputs (parts of) S and optionally Q and P (in packed form), and BD for a given matrix MAT. If the two-step algorithm is used, the routine outputs also O explicitly or in a packed form.

S, Q, P and BD (and also O if the two-step algorithm is selected) may then be used to obtain selected singular vectors with subroutines *bd_inviter2()* or *bd_deflate2()*.

Synopsis:

```
call select_singval_cmp(  mat(:m,:n) , nsing , s(:min(m,n)) , failure ,␣
↪sort=sort , mul_size=mul_size , vector=vector , abstol=abstol , ls=ls ,
↪ theta=theta , d=d(:min(m,n)) , e=e(:min(m,n)) , tauq=tauq(:min(m,n)) ,␣
↪taup=taup(:min(m,n)) , scaling=scaling , init=init , dqds=dqds )

call select_singval_cmp(  mat(:m,:n) , rlmat(:min(m,n),:min(m,n)) , nsing ,
↪ s(:min(m,n)) , failure , sort=sort , mul_size=mul_size , vector=vector␣
↪, abstol=abstol , ls=ls , theta=theta , d=d(:min(m,n)) , e=e(:min(m,n))␣
↪, tauo=tauo(:min(m,n)) , tauq=tauq(:min(m,n)) , taup=taup(:min(m,n)) ,␣
↪scaling=scaling , init=init , dqds=dqds )
```

Examples:

ex1_select_singval_cmp.F90

ex2_select_singval_cmp.F90

select_singval_cmp2()

Purpose:

select_singval_cmp2() computes all or some of the greatest singular values of a real m-by-n matrix MAT.

The Singular Value decomposition (SVD) is written:

$$MAT = U * S * V^T$$

where S is a m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is a m-by-m orthogonal matrix, and V is a n-by-n orthogonal matrix. The diagonal elements of S are the singular values of MAT; they are real and non-negative.

Both one-step and a two-step algorithms are available in **select_singval_cmp2()** for the preliminary reduction of the input matrix MAT to bidiagonal form.

In the one-step algorithm, the original matrix MAT is directly reduced to upper or lower bidiagonal form BD by an orthogonal transformation:

$$Q^T * MAT * P = BD$$

where Q and P are orthogonal (see [Golub_VanLoan:1996] [Lawson_Hanson:1974] [Howell_etal:2008]).

In the two-step algorithm, the original matrix MAT is also reduced to upper bidiagonal form BD, but if:

- m >= n, a QR factorization of the real m-by-n matrix MAT is first computed

  $$MAT = O * R$$

  where O is orthogonal and R is upper triangular. In a second step, the n-by-n upper triangular matrix R is reduced to upper bidiagonal form BD by an orthogonal transformation:

  $$Q^T * R * P = BD$$

  where Q and P are orthogonal and BD is an upper bidiagonal matrix.

- m < n, an LQ factorization of the real m-by-n matrix MAT is first computed

  $$MAT = L * O$$

  where O is orthogonal and L is lower triangular. In a second step, the m-by-m lower triangular matrix L is also reduced to upper bidiagonal form BD by an orthogonal transformation :

  $$Q^T * L * P = BD$$

  where Q and P are orthogonal and BD is an upper bidiagonal matrix.

This two-step reduction algorithm will be more efficient if m is much larger than n or if n is much larger than m.

In both the one-step and two-step algorithms, the singular values S of the bidiagonal matrix BD, which are also the singular values of MAT, are then computed by the DQDS algorithm [Parlett_Marques:2000] [Li_etal:2014] or a bisection algorithm (see [Golub_VanLoan:1996]; Sec.8.5). The bisection method is applied (implicitly) to the associated min(m,n)-by-min(m,n) symmetric tridiagonal matrix

$$BD^T * BD$$

whose eigenvalues are the squares of the singular values of BD by using the differential stationary form of the QD algorithm of Rutishauser (see [Fernando:1998]; Sec.3.1).

Both the DQDS and bisection algorithms are able to compute the singular values of BD to high relative accuracy, bisection is more accurate, but can be much slower than DQDS when all singular values are computed for some bidiagonal matrices. Thus, the default is to use the DQDS algorithm if all the singular values are requested and bisection if only selected singular values are requested, but the user can select the desired algorithm with the optional logical argument *DQDS* in the call to **select_singval_cmp2()**. Note that the DQDS algorithm always computes all the singular values of BD, subset computation of the singular values is only possible with the bisection algorithm.

The routine outputs (parts of) S and optionally Q and P (in packed form), and BD for a given matrix MAT. If the two-step algorithm is used, the routine outputs also O explicitly or in a packed form.

S, Q, P and BD (and also O if the two-step algorithm is selected) may then be used to obtain selected singular vectors with subroutines *bd_inviter2()* or *bd_deflate2()*.

select_singval_cmp2() is faster than *select_singval_cmp()*, but is slightly less accurate if bisection is used to compute the singular values. On the other hand, **select_singval_cmp2()** and *select_singval_cmp()* are equivalent if DQDS is used.

Synopsis:

```
call select_singval_cmp2(  mat(:m,:n) , nsing , s(:min(m,n)) , failure ,␣
↪sort=sort , mul_size=mul_size , vector=vector , abstol=abstol , ls=ls ,
↪ theta=theta , d=d(:min(m,n)) , e=e(:min(m,n)) , tauq=tauq(:min(m,n)) ,␣
↪taup=taup(:min(m,n)) , scaling=scaling , init=init , dqds=dqds )

call select_singval_cmp2(  mat(:m,:n) , rlmat(:min(m,n),:min(m,n)) , nsing␣
↪, s(:min(m,n)) , failure , sort=sort , mul_size=mul_size , vector=vector␣
↪, abstol=abstol , ls=ls , theta=theta , d=d(:min(m,n)) , e=e(:min(m,n))␣
↪, tauo=tauo(:min(m,n)) , tauq=tauq(:min(m,n)) , taup=taup(:min(m,n)) ,␣
↪scaling=scaling , init=init , dqds=dqds )
```

Examples:

ex1_select_singval_cmp2.F90

ex2_select_singval_cmp2.F90

select_singval_cmp3()

Purpose:

select_singval_cmp3() computes all or some of the greatest singular values of a real m-by-n matrix MAT with m>=n.

The Singular Value decomposition (SVD) is written:

$$MAT = U * S * V^T$$

where S is a m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is a m-by-m orthogonal matrix, and V is a n-by-n orthogonal matrix. The diagonal elements of S are the singular values of MAT; they are real and non-negative.

Both one-step and a two-step algorithms are available in **select_singval_cmp3()** for the preliminary reduction of the input matrix MAT to bidiagonal form.

In the one-step algorithm, the original matrix MAT is directly reduced to upper bidiagonal form BD by an orthogonal transformation:

$$Q^T * MAT * P = BD$$

where Q and P are orthogonal (see [Lawson_Hanson:1974] [Golub_VanLoan:1996]). The fast Ralha-Barlow one-sided method is used for this purpose (see [Ralha:2003] [Barlow_etal:2005] [Bosner_Barlow:2007]).

In the two-step algorithm, the original matrix MAT is also reduced to upper bidiagonal form BD. But, a QR factorization of the real m-by-n matrix MAT is first computed

$$MAT = O * R$$

where O is orthogonal and R is upper triangular. In a second step, the n-by-n upper triangular matrix R is reduced to upper bidiagonal form BD by an orthogonal transformation:

$$Q^T * R * P = BD$$

where Q and P are orthogonal and BD is an upper bidiagonal matrix. The fast Ralha-Barlow one-sided method is also used for this purpose (see [Ralha:2003] [Barlow_etal:2005] [Bosner_Barlow:2007]).

This two-step reduction algorithm will be more efficient if m is much larger than n.

In both the one-step and two-step algorithms, the singular values S of the bidiagonal n-by-n matrix BD, which are also the singular values of MAT, are then computed by the DQDS algorithm [Parlett_Marques:2000] [Li_etal:2014] or a bisection algorithm applied to the Tridiagonal Golub-Kahan (TGK) form of the bidiagonal matrix BD (see [Fernando:1998]; Sec.3.3).

Both the DQDS and bisection algorithms are able to compute the singular values of BD to high relative accuracy, bisection is more accurate, but can be much slower than DQDS when all singular values are computed for some bidiagonal matrices. Thus, the default is to use the DQDS algorithm if all the singular values are requested and to use bisection if only selected singular values are requested, but the user can select the desired algorithm with the optional logical argument *DQDS* in the call to **select_singval_cmp3()**. Note that the DQDS algorithm always computes all the singular values of BD, subset computation of the singular values is only possible with the bisection algorithm.

The routine outputs (parts of) S, Q and optionally P (in packed form) and BD for a given matrix MAT. If the two-step algorithm is used, the routine outputs also O explicitly or in a packed form.

S, Q, P and BD (and also O if the two-step algorithm is selected) may then be used to obtain selected singular vectors with subroutines *bd_inviter2()* or *bd_deflate2()*.

In many cases, **select_singval_cmp3()** is faster than *select_singval_cmp()*, but is slightly less accurate, due to the use of the Ralha-Barlow one-sided bidiagonal reduction algorithm. However, in a parallel setting (e.g., when OpenMP is used), **select_singval_cmp3()** will be signicantly faster than *select_singval_cmp()* only when the number of available processors is relatively small (e.g., less than 16 roughly). The reverse can be observed when a large number of processors is available.

Synopsis:

```
call select_singval_cmp3( mat(:m,:n) , nsing , s(:n) , failure , sort=sort ,
↪ mul_size=mul_size , vector=vector , abstol=abstol , ls=ls , theta=theta ,␣
↪d=d(:n) , e=e(:n) , p=p(:n,:n) , gen_p=gen_p , scaling=scaling , init=init ,
↪ dqds=dqds , failure_bd=failure_bd )

call select_singval_cmp3( mat(:m,:n) , rlmat(:min(m,n),:min(m,n)) , nsing␣
↪, s(:n) , failure , sort=sort , mul_size=mul_size , vector=vector ,␣
↪abstol=abstol , ls=ls , theta=theta , d=d(:n) , e=e(:n) , tauo=tauo(:min(m,
↪n)) , p=p(:n,:n) , gen_p=gen_p , scaling=scaling , init=init , dqds=dqds ,␣
↪failure_bd=failure_bd )
```

Examples:

ex1_select_singval_cmp3.F90

ex1_select_singval_cmp3_bis.F90

ex2_select_singval_cmp3.F90

ex2_select_singval_cmp3_bis.F90

select_singval_cmp4()

Purpose:

select_singval_cmp4() computes all or some of the greatest singular values of a real m-by-n matrix MAT with m>=n.

The Singular Value decomposition (SVD) is written:

$$MAT = U * S * V^T$$

where S is a m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is a m-by-m orthogonal matrix, and V is a n-by-n orthogonal matrix. The diagonal elements of S are the singular values of MAT; they are real and non-negative.

Both one-step and a two-step algorithms are available in **select_singval_cmp4()** for the preliminary reduction of the input matrix MAT to bidiagonal form.

In the one-step algorithm, the original matrix MAT is directly reduced to upper bidiagonal form BD by an orthogonal transformation:

$$Q^T * MAT * P = BD$$

where Q and P are orthogonal (see [Lawson_Hanson:1974] [Golub_VanLoan:1996]). The fast Ralha-Barlow one-sided method is used for this purpose (see [Ralha:2003] [Barlow_etal:2005] [Bosner_Barlow:2007]).

In the two-step algorithm, the original matrix MAT is also reduced to upper bidiagonal form BD. But, a QR factorization of the real m-by-n matrix MAT is first computed

$$MAT = O * R$$

where O is orthogonal and R is upper triangular. In a second step, the n-by-n upper triangular matrix R is reduced to upper bidiagonal form BD by an orthogonal transformation:

$$Q^T * R * P = BD$$

where Q and P are orthogonal and BD is an upper bidiagonal matrix. The fast Ralha-Barlow one-sided method is also used for this purpose (see [Ralha:2003] [Barlow_etal:2005] [Bosner_Barlow:2007]).

This two-step reduction algorithm will be more efficient if m is much larger than n.

In both the one-step and two-step algorithms, the singular values S of the bidiagonal matrix BD, which are also the singular values of MAT, are then computed by the DQDS algorithm [Parlett_Marques:2000] [Li_etal:2014] or a bisection algorithm (see [Golub_VanLoan:1996]; Sec.8.5). The bisection method is applied (implicitly) to the associated n-by-n symmetric tridiagonal matrix

$$BD^T * BD$$

whose eigenvalues are the squares of the singular values of BD by using the differential stationary form of the QD algorithm of Rutishauser (see [Fernando:1998]; Sec.3.1).

Both the DQDS and bisection algorithms are able to compute the singular values of BD to high relative accuracy, bisection is more accurate, but can be much slower than DQDS when all singular values are computed for some bidiagonal matrices. Thus, the default is to use the DQDS algorithm if all the singular values are requested and bisection if only selected singular values are requested, but the user can select the desired algorithm with the optional logical argument *DQDS* in the call to **select_singval_cmp4()**. Note that the DQDS algorithm always computes all the singular values of BD, subset computation of the singular values is only possible with the bisection algorithm.

The routine outputs (parts of) S, Q and optionally P (in packed form) and BD for a given matrix MAT. If the two-step algorithm is used, the routine outputs also O explicitly or in a packed form.

S, Q, P and BD (and also O if the two-step algorithm is selected) may then be used to obtain selected singular vectors with subroutines *bd_inviter2()* or *bd_deflate2()*.

select_singval_cmp4() is faster than *select_singval_cmp3()*, but is slightly less accurate if bisection is used to compute the singular values. On the other hand, **select_singval_cmp4()** and *select_singval_cmp3()* are equivalent if DQDS is used.

In many cases, **select_singval_cmp4()** is also faster than *select_singval_cmp2()*, but is slightly less accurate, due to the use of the Ralha-Barlow one-sided bidiagonal reduction algorithm. However, in a parallel setting (e.g., when OpenMP is used), **select_singval_cmp4()** will be signicantly faster than *select_singval_cmp2()* only when the number of available processors is relatively small (e.g., less than 16 roughly). The reverse can be observed when a large number of processors is available.

Synopsis:

```
call select_singval_cmp4(  mat(:m,:n) , nsing , s(:n) , failure , sort=sort ,
↪ mul_size=mul_size , vector=vector , abstol=abstol , ls=ls , theta=theta ,␣
↪d=d(:n) , e=e(:n) , p=p(:n,:n) , gen_p=gen_p , scaling=scaling , init=init ,
↪ dqds=dqds , failure_bd=failure_bd )

call select_singval_cmp4(  mat(:m,:n) , rlmat(:min(m,n),:min(m,n)) , nsing␣
↪, s(:n) , failure , sort=sort , mul_size=mul_size , vector=vector ,␣
↪abstol=abstol , ls=ls , theta=theta , d=d(:n) , e=e(:n) , tauo=tauo(:min(m,
↪n)) , p=p(:n,:n) , gen_p=gen_p , scaling=scaling , init=init , dqds=dqds ,␣
↪failure_bd=failure_bd )
```

Examples:

ex1_select_singval_cmp4.F90

ex1_select_singval_cmp4_bis.F90

ex2_select_singval_cmp4.F90

ex2_select_singval_cmp4_bis.F90

singval_sort()

Purpose:

Given the singular values as output from *bd_svd()*, *bd_svd2()*, *svd_cmp()*, *svd_cmp2()*, *svd_cmp3()*, svd_cm4(), svd_cm5(), svd_cm7() or *svd_cmp8()*, **singval_sort()** sorts the singular values into ascending or descending order.

Synopsis:

```
call singval_sort( sort , d(:n) )
```

singvec_sort()

Purpose:

Given the singular values and (left or right) vectors as output from *bd_svd()*, *bd_svd2()*, *svd_cmp()*, *svd_cmp2()*, *svd_cmp3()*, svd_cm4(), svd_cm5(), svd_cm6(),, svd_cm7() or *svd_cmp8()*, **singvec_sort()** sorts the singular values into ascending or descending order and reorders the associated singular vectors accordingly.

Synopsis:

```
call singvec_sort( sort , d(:n) , u(:,:n) )
```

svd_sort()

Purpose:

Given the singular values and the associated left and right singular vectors as output from *bd_svd()*, *svd_cmp()*, *svd_cmp3()*, svd_cm4(), svd_cm5(), svd_cm6(),, svd_cm7() or *svd_cmp8()*, **svd_sort()** sorts the singular values into ascending or descending order, and, rearranges the left and right singular vectors correspondingly.

Synopsis:

```
call svd_sort( sort , d(:n) , u(:,:n) , v(:,:n) )
call svd_sort( sort , d(:n) , u(:,:n)           )
call svd_sort( sort , d(:n)                     )
```

svd_sort2()

Purpose:

Given the singular values and the associated left and right singular vectors as output from *bd_svd2()* or *svd_cmp2()*, **svd_sort2()** sorts the singular values into ascending or descending order, and, rearranges the left and right singular vectors correspondingly.

Synopsis:

```
call svd_sort2( sort , d(:n) , u(:,:n) , vt(:n,:) )
call svd_sort2( sort , d(:n) , u(:,:n)            )
call svd_sort2( sort , d(:n)                      )
```

maxdiag_gkinv_qr()

Purpose:

maxdiag_gkinv_qr() computes the index of the element of maximum absolute value in the diagonal entries of

$$(GK - lambda * I)^{-1}$$

where GK is a n-by-n symmetric tridiagonal matrix with a zero diagonal, I is the identity matrix and lambda is a scalar.

The diagonal entries of (GK - lambda * I)$^{-1}$ are computed by means of the QR factorization of GK - lambda * I.

For more details, see [Bini_etal:2005].

It is assumed that GK is unreduced, but no check is done in the subroutine to verify this assumption.

Synopsis:

```
maxdiag_gkinv = maxdiag_gkinv_qr( e(:) , lambda )
```

maxdiag_gkinv_ldu()

Purpose:

maxdiag_gkinv_ldu() computes the index of the element of maximum absolute value in the diagonal entries of

$$(GK - lambda * I)^{-1}$$

where GK is a n-by-n symmetric tridiagonal matrix with a zero diagonal, I is the identity matrix and lambda is a scalar.

The diagonal entries of (GK - lambda * I)$^{-1}$ are computed by means of LDU and UDL factorizations of GK - lambda * I.

For more details, see [Fernando:1997].

It is assumed that GK is unreduced, but no check is done in the subroutine to verify this assumption.

Synopsis:

```
maxdiag_gkinv = maxdiag_gkinv_ldu( e(:) , lambda )
```

gk_qr_cmp()

Purpose:

gk_qr_cmp() factorizes the symmetric matrix GK - lambda * I, where GK is a n-by-n symmetric tridiagonal matrix with a zero diagonal, I is the identity matrix and lambda is a scalar. as

$$GK - lambda * I = Q * R$$

where Q is an orthogonal matrix represented as the product of n-1 Givens rotations and R is an upper triangular matrix with at most two non-zero super-diagonal elements per column.

The parameter lambda is included in the routine so that **gk_qr_cmp()** may be used to obtain eigenvectors of GK by inverse iteration.

The subroutine also computes the index of the entry of maximum absolute value in the diagonal of (GK - lambda * I)$^{-1}$, which provides a good initial approximation to start the inverse iteration process for computing the eigenvector associated with the eigenvalue lambda.

For further details, see [Bini_etal:2005] [Fernando:1997] [Parlett_Dhillon:1997].

Synopsis:

```
call gk_qr_cmp( e(:n-1) , lambda , cs(:n-1) , sn(:n-1) , diag(:n) , sup1(:n) ,
↪ sup2(:n) , maxdiag_gkinv )
```

bd_inviter()

Purpose:

bd_inviter() computes the left and right singular vectors of a real n-by-n bidiagonal matrix BD corresponding to specified singular values, using Fernando's method and inverse iteration on the Tridiagonal Golub-Kahan (TGK) form of the bidiagonal matrix BD.

The singular values used as input of **bd_inviter()** can be computed with a call to *bd_svd()*, *bd_singval()*, *bd_singval2()*, *bd_lasq1()* or *bd_dqds()*.

Moreover, the singular values used as input of **bd_inviter()** can be computed to high accuracy for getting more robust results. This will be the case if the optional parameter *ABSTOL* is used and set to `sqrt(lamch("S"))`, or equivalently to `sqrt(safmin)` (where `safmin` is a public numerical constant exported by the *Num_Constants* module), in the preliminary calls to *bd_singval()* or *bd_singval2()*, which compute the singular values of `BD` by bisection. Alternatively, highly accurate singular values can be computed by the DQDS algorithm (e.g., if optional parameter *DQDS* is used and set to `true` in the calls to *bd_singval()* or *bd_singval2()*, or, if *bd_lasq1()* or *bd_dqds()* are directly used to compute the singular values of `BD` by the DQDS algorithm).

See description of *bd_singval()*, *bd_singval2()*, *bd_lasq1()* or *bd_dqds()* for more details.

Synopsis:

```
call bd_inviter( upper , d(:n) , e(:n) , s     , leftvec(:n)    ,␣
↪rightvec(:n)    , failure , maxiter=maxiter , scaling=scaling ,␣
↪initvec=initvec )

call bd_inviter( upper , d(:n) , e(:n) , s(:p) , leftvec(:n,:p) , rightvec(:n,
↪:p) , failure , maxiter=maxiter , ortho=ortho , backward_sweep=backward_
↪sweep , scaling=scaling , initvec=initvec )
```

Examples:

ex1_bd_inviter.F90

ex2_bd_inviter.F90

bd_inviter2()

Purpose:

bd_inviter2() computes the left and right singular vectors of a full real `m`-by-`n` matrix `MAT` corresponding to specified singular values, using inverse iteration.

It is required that the original matrix `MAT` has been first reduced to upper or lower bidiagonal form `BD` by an orthogonal transformation:

$$Q^T * MAT * P = BD$$

where `Q` and `P` are orthogonal, and that selected singular values of `BD` have been computed.

These first steps can be performed in several ways:

- with a call to subroutine *select_singval_cmp()* or *select_singval_cmp2()* (with parameters *D*, *E*, *TAUQ* and *TAUP*), which reduce the input matrix `MAT` with an one-step Golub-Kahan bidiagonalization algorithm and compute (selected) singular values of the resulting bidiagonal matrix `BD`;
- with a call to subroutine *select_singval_cmp()* or *select_singval_cmp2()* (with parameters *D*, *E*, *TAUQ*, *TAUP*, *RLMAT* and *TAUO*), which reduce the input matrix `MAT` with a two-step Golub-Kahan bidiagonalization algorithm and compute (selected) singular values of the resulting bidiagonal matrix `BD`;
- using first, an one-step bidiagonalization Golub-Kahan algorithm, with a call to *bd_cmp()* (with parameters *D*, *E*, *TAUQ* and *TAUP*) followed by a call to *bd_svd()*, *bd_singval()* or *bd_singval2()* for computing (selected) singular values of the resulting bidiagonal matrix `BD` (this is equivalent of using subroutines *select_singval_cmp()* or *select_singval_cmp2()* above with parameters *D*, *E*, *TAUQ* and *TAUP*);
- using first, a two-step bidiagonalization Golub-Kahan algorithm, with a call to *bd_cmp()* (with parameters *D*, *E*, *TAUQ*, *TAUP*, *RLMAT* and *TAUO*) followed by a call to *bd_svd()*, *bd_singval()* or *bd_singval2()* for computing (selected) singular values of the resulting bidiagonal matrix `BD` (this is equivalent of using subroutines *select_singval_cmp()* or *select_singval_cmp2()* above with parameters *D*, *E*, *TAUQ*, *TAUP*, *RLMAT* and *TAUO*).

If `m>=n`, these first steps can also be performed:

- with a call to subroutine *select_singval_cmp3()* or *select_singval_cmp4()* (with parameters *D*, *E* and *P*), which reduce the input matrix MAT with an one-step Ralha-Barlow bidiagonalization algorithm and compute (selected) singular values of the resulting bidiagonal matrix BD.
- with a call to subroutine *select_singval_cmp3()* or *select_singval_cmp4()* (with parameters *D*, *E*, *P*, *RLMAT* and *TAUO*), which reduce the input matrix MAT with a two-step Ralha-Barlow bidiagonalization algorithm and compute (selected) singular values of the resulting bidiagonal matrix BD.
- using first, a one-step Ralha-Barlow bidiagonalization algorithm, with a call to *bd_cmp2()* (with parameters *D*, *E* and *P*), followed by a call to *bd_svd()*, *bd_singval()* or *bd_singval2()* for computing (selected) singular values of the resulting bidiagonal matrix BD (this is equivalent of using subroutines *select_singval_cmp3()* or *select_singval_cmp4()* above with parameters *D*, *E* and *P*).

If max(m,n) is much larger than min(m,n), it is more efficient to use a two-step bidiagonalization algorithm than a one-step algorithm. Moreover, if m>=n, using the Ralha-Barlow bidiagonalization method will be a faster method in serial mode or if the number of available processors is relatively small in parallel mode (e.g., when OpenMP is used).

Once (selected) singular values of BD, which are also singular values of MAT, have been computed by using one of the above paths, a call to **bd_inviter2()** will compute the associated singular vectors of MAT if the appropriate parameters *TAUQ*, *TAUP*, *P*, *TAUO*, *RLMAT* and *TAUO* (depending on the previous selected path) are specified in the call to **bd_inviter2()**.

Moreover, independently of the selected paths, note that the singular values used as input of **bd_inviter2()** can be computed to high accuracy for getting more robust results. This will be the case if the optional parameter *ABSTOL* is used and set to sqrt(lamch("S")), or equivalently to sqrt(safmin) (where safmin is a public numerical constant exported by the *Num_Constants* module)) in the preliminary calls to *select_singval_cmp()*, *select_singval_cmp2()*, *select_singval_cmp3()*, *select_singval_cmp4()*, *bd_singval()* or *bd_singval2()*, which compute the singular values of BD (and MAT) by bisection. Alternatively, highly accurate singular values can also be estimated with the DQDS algorithm (e.g., if optional parameter *DQDS* is used and set to true in the preliminary calls to *select_singval_cmp()*, *select_singval_cmp2()*, *select_singval_cmp3()*, *select_singval_cmp4()*, *bd_singval()*, *bd_singval2()* or if *bd_lasq1()* or *bd_dqds()* are directly used to compute the singular values of BD by the DQDS algorithm).

See, for example, the description of *select_singval_cmp()* or *select_singval_cmp3()* for more details.

Synopsis:

```
call bd_inviter2( mat(:m,:n) , tauq(:min(m,n)) , taup(:min(m,n)) ,␣
↪d(:min(m,n)) , e(:min(m,n)) , s(:p) , leftvec(:m,:p) , rightvec(:n,:p) ,␣
↪failure, maxiter=maxiter , ortho=ortho , backward_sweep=backward_sweep ,␣
↪scaling=scaling , initvec=initvec )

call bd_inviter2( mat(:m,:n) , p(:n,:n) , d(:n) , e(:n) , s(:p) , leftvec(:m,
↪:p) , rightvec(:n,:p) , failure, maxiter=maxiter , ortho=ortho , backward_
↪sweep=backward_sweep , scaling=scaling , initvec=initvec )

call bd_inviter2( mat(:m,:n) , tauq(:min(m,n)) , taup(:min(m,n)) ,␣
↪rlmat(:min(m,n),:min(m,n)) , d(:min(m,n)) , e(:min(m,n)) , s(:p) ,
↪ leftvec(:m,:p) , rightvec(:n,:p) , failure, tauo=tauo(:min(m,n))␣
↪, maxiter=maxiter , ortho=ortho , backward_sweep=backward_sweep ,␣
↪scaling=scaling , initvec=initvec )

call bd_inviter2( mat(:m,:n) , rlmat(:min(m,n),:min(m,n)) , p(:n,:n) ,
↪ d(:n) , e(:n) , s(:p) , leftvec(:m,:p) , rightvec(:n,:p) , failure,
↪ tauo=tauo(:min(m,n)) , maxiter=maxiter , ortho=ortho , backward_
↪sweep=backward_sweep , scaling=scaling , initvec=initvec )
```

Examples:

ex1_bd_inviter2.F90

ex1_bd_inviter2_bis.F90

ex2_bd_inviter2.F90

ex1_select_singval_cmp.F90

ex1_select_singval_cmp2.F90

ex1_select_singval_cmp3.F90

ex1_select_singval_cmp3_bis.F90

ex1_select_singval_cmp4.F90

ex1_select_singval_cmp4_bis.F90

upper_bd_dsqd()

Purpose:

upper_bd_dsqd() computes:

- the $L * D * L^T$ factorization of the matrix $BD^T * BD - shift * I$, if *FLIP*=`false`;
- the $U * D * U^T$ factorization of the matrix $BD^T * BD - shift * I$, if *FLIP*=`true`;

for a `n`-by-`n` (upper) bidiagonal matrix `BD` and a given scalar shift. `L` and `U` are, respectively, unit lower and unit upper bidiagonal matrices and `D` is a diagonal matrix.

The differential form of the stationary QD algorithm of Rutishauser is used to compute the factorization. See [Fernando:1998] for further details.

The subroutine outputs the diagonal matrix `D` of the factorization, the off-diagonal entries of `L` (or of `U` if *FLIP*=`true`) and the auxiliary variable `T` in the differential form of the stationary QD algorithm.

Synopsis:

```
call upper_bd_dsqd( a(:n) , b(:n-1) , shift , flip , d(:n)                    )
call upper_bd_dsqd( a(:n) , b(:n-1) , shift , flip , d(:n) , t(:n)            )
call upper_bd_dsqd( a(:n) , b(:n-1) , shift , flip , d(:n) , t(:n) , l(:n-1) )
```

upper_bd_dpqd()

Purpose:

upper_bd_dpqd() computes:

- the $L * D * L^T$ factorization of the matrix $BD^T * BD - shift * I$, if *FLIP*=`false`;
- the $U * D * U^T$ factorization of the matrix $BD^T * BD - shift * I$, if *FLIP*=`true`;

for a `n`-by-`n` (upper) bidiagonal matrix `BD` and a given scalar shift. `L` and `U` are, respectively, unit lower and unit upper bidiagonal matrices and `D` is a diagonal matrix.

The differential form of the progressive QD algorithm of Rutishauser is used to compute the factorization (see [Fernando:1998] for further details).

The subroutine outputs the diagonal matrix `D` of the factorization, the off-diagonal entries of `L` (or of `U` if *FLIP*=`true`) and the auxiliary variable `S` in the differential form of the progressive QD algorithm.

Synopsis:

```
call upper_bd_dpqd( a(:n) , b(:n-1) , shift , flip , d(:n)                   )
call upper_bd_dpqd( a(:n) , b(:n-1) , shift , flip , d(:n) , s(:n)           )
call upper_bd_dpqd( a(:n) , b(:n-1) , shift , flip , d(:n) , s(:n) , l(:n-1) )
```

upper_bd_dsqd2()

Purpose:

upper_bd_dsqd2() computes:

- the $L * D * L^T$ factorization of the matrix $BD^T * BD - shift * I$, if *FLIP*=`false`;
- the $U * D * U^T$ factorization of the matrix $BD^T * BD - shift * I$, if *FLIP*=`true`;

for a `n`-by-`n` (upper) bidiagonal matrix `BD` and a given scalar shift. `L` and `U` are, respectively, unit lower and unit upper bidiagonal matrices and `D` is a diagonal matrix.

The differential form of the stationary QD algorithm of Rutishauser is used to compute the factorization from the squared elements of the bidiagonal matrix `BD`. See [Fernando:1998] for further details.

The subroutine outputs the diagonal matrix `D` of the factorization and the auxiliary variable `T` (at the user option) in the differential form of the stationary QD algorithm.

Synopsis:

```
call upper_bd_dsqd2( q2(:n) , e2(:n-1) , shift , flip , d(:n)          )
call upper_bd_dsqd2( q2(:n) , e2(:n-1) , shift , flip , d(:n) , t(:n) )
```

upper_bd_dpqd2()

Purpose:

upper_bd_dpqd2() computes:

- the $L * D * L^T$ factorization of the matrix $BD^T * BD - shift * I$, if *FLIP*=`false`;
- the $U * D * U^T$ factorization of the matrix $BD^T * BD - shift * I$, if *FLIP*=`true`;

for a `n`-by-`n` (upper) bidiagonal matrix `BD` and a given scalar shift. `L` and `U` are, respectively, unit lower and unit upper bidiagonal matrices and `D` is a diagonal matrix.

The differential form of the progressive QD algorithm of Rutishauser is used to compute the factorization from the squared elements of the bidiagonal matrix `BD`. See [Fernando:1998] for further details.

The subroutine outputs the diagonal matrix `D` of the factorization and the auxiliary variable `S` in the differential form of the progressive QD algorithm.

Synopsis:

```
call upper_bd_dpqd2( q2(:n) , e2(:n-1) , shift , flip , d(:n)          )
call upper_bd_dpqd2( q2(:n) , e2(:n-1) , shift , flip , d(:n) , s(:n) )
```

dflgen_bd()

Purpose:

dflgen_bd() computes deflation parameters (e.g., two chains of Givens rotations) for a `n`-by-`n` (upper) bidiagonal matrix `BD` and a given singular value of `BD`.

On output, the arguments *CS_LEFT*, *SN_LEFT*, *CS_RIGHT* and *SN_RIGHT* contain, respectively, the vectors of the cosines and sines coefficients of the chain of `n-1` planar rotations that deflates the real `n`-by-`n` bidiagonal matrix `BD` corresponding to a singular value *LAMBDA*.

For further details, see [Godunov_etal:1993] [Malyshev:2000].

Synopsis:

```
call dflgen_bd( d(:n) , e(:n-1) , lambda , cs_left(:n-1) , sn_left(:n-1) , cs_
↪right(:n-1) , sn_right(:n-1) , scaling=scaling )
```

dflgen2_bd()

Purpose:

dflgen2_bd() computes and applies deflation parameters (e.g., two chains of Givens rotations) for a n-by-n (upper) bidiagonal matrix BD and a given singular value of BD.

On input:

> The arguments *D* and *E* contain, respectively, the main diagonal and off-diagonal of the bidiagonal matrix, and the argument *LAMBDA* contains an estimate of the singular value.

On output:

> The arguments *D* and *E* contain, respectively, the new main diagonal and off-diagonal of the deflated bidiagonal matrix if the argument *DEFLATE* is set to true, otherwise *D* and *E* are not changed.
>
> The arguments *CS_LEFT*, *SN_LEFT*, *CS_RIGHT* and *SN_RIGHT* contain, respectively, the vectors of the cosines and sines coefficients of the chain of n-1 planar rotations that deflates the real n-by-n bidiagonal matrix BD corresponding to the singular value *LAMBDA*. One chain is applied to the left of BD (*CS_LEFT*, *SN_LEFT*) and the other is applied to the right of BD (*CS_RIGHT*, *SN_RIGHT*).

For further details, see [Godunov_etal:1993] [Malyshev:2000].

Synopsis:

```
call dflgen2_bd( d(:n) , e(:n-1) , lambda , cs_left(:n-1) , sn_left(:n-1) ,␣
↪cs_right(:n-1) , sn_right(:n-1) , deflate , scaling=scaling )
```

dflapp_bd()

Purpose:

dflapp_bd() deflates a real n-by-n (upper) bidiagonal matrix BD by two chains of planar rotations produced by *dflgen_bd()* or *dflgen2_bd()*.

On entry, the arguments *D* and *E* contain, respectively, the main diagonal and off-diagonal of the bidiagonal matrix.

On output, the arguments *D* and *E* contain, respectively, the new main diagonal and off-diagonal of the deflated bidiagonal matrix if the argument *DEFLATE* is set to true on output of **dflapp_bd()**.

For further details, see [Godunov_etal:1993] [Malyshev:2000].

Synopsis:

```
call dflapp_bd( d(:n) , e(:n-1) , cs_left(:n-1) , sn_left(:n-1) , cs_right(:n-
↪1) , sn_right(:n-1) , deflate )
```

qrstep_bd()

Purpose:

qrstep_bd() performs one QR step with a given shift *LAMBDA* on a n-by-n real (upper) bidiagonal matrix BD.

On entry, the arguments *D* and *E* contain, respectively, the main diagonal and off-diagonal of the bidiagonal matrix.

On output, the arguments *D* and *E* contain, respectively, the new main diagonal and off-diagonal of the deflated bidiagonal matrix if the logical argument *DEFLATE* is set to true on exit or if the optional logical argument *UPDATE_BD* is used with the value true on entry; otherwise the arguments *D* and *E* are not modified.

The two chains of n-1 planar rotations produced during the QR step are saved in the arguments *CS_LEFT*, *SN_LEFT*, *CS_RIGHT*, *SN_RIGHT*.

For further details, see [Mastronardi_etal:2006].

Synopsis:

```
call qrstep_bd( d(:n) , e(:n-1) , lambda , cs_left(:n-1) , sn_left(:n-1) , cs_
↪right(:n-1) , sn_right(:n-1) , deflate, update_bd )
```

qrstep_zero_bd()

Purpose:

qrstep_zero_bd() performs one implicit QR step with a zero shift on a n-by-n real (upper) bidiagonal matrix BD.

On entry, the arguments *D* and *E* contain, respectively, the main diagonal and off-diagonal of the bidiagonal matrix.

On output, the arguments *D* and *E* contain, respectively, the new main diagonal and off-diagonal of the deflated bidiagonal matrix if the logical argument *DEFLATE* is set to true on exit or if the optional logical argument *UPDATE_BD* is used with the value true on entry; otherwise the arguments *D* and *E* are not modified.

The two chains of n-1 planar rotations produced during the QR step are saved in the arguments *CS_LEFT*, *SN_LEFT*, *CS_RIGHT*, *SN_RIGHT*.

For further details, see [Demmel_Kahan:1990].

Synopsis:

```
call qrstep_zero_bd( d(:n) , e(:n-1) , cs_left(:n-1) , sn_left(:n-1) , cs_
↪right(:n-1) , sn_right(:n-1) , deflate, update_bd )
```

upper_bd_deflate()

Purpose:

upper_bd_deflate() computes the left and right singular vectors of a real (upper) bidiagonal matrix BD corresponding to specified singular values, using a deflation technique on the BD matrix.

upper_bd_deflate() is a low-level subroutine used by *bd_deflate()* and *bd_deflate2()* subroutines. Its use as a stand-alone method for computing singular vectors of a bidiagonal matrix is not recommended.

Note also that the sign of the singular vectors computed by **upper_bd_deflate**() is arbitrary and not necessarily consistent between the left and right singular vectors. In order to compute consistent singular triplets, subroutine *bd_deflate()* must be used instead.

Synopsis:

```
call upper_bd_deflate( d(:n) , e(:n-1) , singval     , leftvec(:n)    ,␣
↪rightvec(:n)    , failure , max_qr_steps=max_qr_steps , scaling=scaling )
call upper_bd_deflate( d(:n) , e(:n-1) , singval(:p) , leftvec(:n,:p) ,␣
↪rightvec(:n,:p) , failure , max_qr_steps=max_qr_steps , scaling=scaling )
```

bd_deflate()

Purpose:

bd_deflate() computes the left and right singular vectors of a real n-by-n bidiagonal matrix BD corresponding to specified singular values, using deflation techniques on the bidiagonal matrix BD.

It is highly recommended that the singular values used as input of **bd_deflate**() have been computed to high accuracy for getting more robust results. This will be the case if the optional parameter *ABSTOL* is used and set to sqrt(lamch("S")), or equivalently to sqrt(safmin) (where safmin is a public numerical constant exported by the *Num_Constants* module), in the preliminary calls to *bd_singval()* or *bd_singval2()*, which compute the singular values of BD by bisection. Alternatively, highly accurate singular values can be computed by the DQDS algorithm (e.g., if optional parameter *DQDS* is used and set to true in the calls to *bd_singval()* or *bd_singval2()*, or, if *bd_lasq1()* or *bd_dqds()* are directly used to compute the singular values of BD by the DQDS algorithm).

See description of *bd_singval()*, *bd_singval2()*, *bd_lasq1()* or *bd_dqds()* for more details.

Synopsis:

```
call bd_deflate( upper , d(:n) , e(:n) , s(:p) , leftvec(:n,:p) , rightvec(:n,
↪:p) , failure , max_qr_steps=max_qr_steps , ortho=ortho , scaling=scaling ,␣
↪inviter=inviter )
```

Examples:

ex1_bd_deflate.F90

bd_deflate2()

Purpose:

bd_deflate2() computes the left and right singular vectors of a full real m-by-n matrix MAT corresponding to specified singular values, using deflation techniques.

It is required that the original matrix MAT has been first reduced to upper or lower bidiagonal form BD by an orthogonal transformation:

$$Q^T * MAT * P = BD$$

where Q and P are orthogonal, and that selected singular values of BD have been computed.

These first steps can be performed in several ways:

- with a call to subroutine *select_singval_cmp()* or *select_singval_cmp2()* (with parameters *D*, *E*, *TAUQ* and *TAUP*), which reduce the input matrix MAT with an one-step Golub-Kahan bidiagonalization algorithm and compute (selected) singular values of the resulting bidiagonal matrix BD;
- with a call to subroutine *select_singval_cmp()* or *select_singval_cmp2()* (with parameters *D*, *E*, *TAUQ*, *TAUP*, *RLMAT* and *TAUO*), which reduce the input matrix MAT with a two-step Golub-Kahan bidiagonalization algorithm and compute (selected) singular values of the resulting bidiagonal matrix BD;
- using first, an one-step bidiagonalization Golub-Kahan algorithm, with a call to *bd_cmp()* (with parameters *D*, *E*, *TAUQ* and *TAUP*) followed by a call to *bd_svd()*, *bd_singval()* or *bd_singval2()* for computing (selected) singular values of the resulting bidiagonal matrix BD (this is equivalent of using subroutines *select_singval_cmp()* or *select_singval_cmp2()* above with parameters *D*, *E*, *TAUQ* and *TAUP*);
- using first, a two-step bidiagonalization Golub-Kahan algorithm, with a call to *bd_cmp()* (with parameters *D*, *E*, *TAUQ*, *TAUP*, *RLMAT* and *TAUO*) followed by a call to *bd_svd()*, *bd_singval()* or *bd_singval2()* for computing (selected) singular values of the resulting bidiagonal matrix BD (this is equivalent of using subroutines *select_singval_cmp()* or *select_singval_cmp2()* above with parameters *D*, *E*, *TAUQ*, *TAUP*, *RLMAT* and *TAUO*).

If m>=n, these first steps can also be performed:

- with a call to subroutine *select_singval_cmp3()* or *select_singval_cmp4()* (with parameters *D*, *E* and *P*), which reduce the input matrix MAT with an one-step Ralha-Barlow bidiagonalization algorithm and compute (selected) singular values of the resulting bidiagonal matrix BD.
- with a call to subroutine *select_singval_cmp3()* or *select_singval_cmp4()* (with parameters *D*, *E*, *P*, *RLMAT* and *TAUO*), which reduce the input matrix MAT with a two-step Ralha-Barlow bidiagonalization algorithm and compute (selected) singular values of the resulting bidiagonal matrix BD.
- using first, a one-step Ralha-Barlow bidiagonalization algorithm, with a call to *bd_cmp2()* (with parameters *D*, *E* and *P*), followed by a call to *bd_svd()*, *bd_singval()* or *bd_singval2()* for computing (selected) singular values of the resulting bidiagonal matrix BD (this is equivalent of using subroutines *select_singval_cmp3()* or *select_singval_cmp4()* above with parameters *D*, *E* and *P*).

If max(m,n) is much larger than min(m,n), it is more efficient to use a two-step bidiagonalization algorithm than a one-step algorithm. Moreover, if m>=n, using the Ralha-Barlow bidiagonalization method will be a faster method.

Once (selected) singular values of BD, which are also singular values of MAT, have been computed by using one of the above paths, a call to **bd_deflate2()** will compute the associated singular vectors of MAT if the appropriate parameters *TAUQ*, *TAUP*, *P*, *TAUO*, *RLMAT* and *TAUO* (depending on the previous selected path) are specified in the call to **bd_deflate2()**.

Moreover, independently of the selected paths, it is also highly recommended that the singular values used as input of **bd_deflate2()** have been computed to high accuracy for geting more robust results. This will be the case if the optional parameter *ABSTOL* is used and set to `sqrt(lamch("S"))`, or equivalently to `sqrt(safmin)` (where `safmin` is a public numerical constant exported by the *Num_Constants* module)) in the preliminary calls to *select_singval_cmp()*, *select_singval_cmp2()*, *select_singval_cmp3()*, *select_singval_cmp4()*, *bd_singval()* or *bd_singval2()*, which compute the singular values of BD (and MAT) by bisection. Alternatively, highly accurate singular values can also be estimated with the DQDS algorithm (e.g., if optional parameter *DQDS* is used and set to `true` in the preliminary calls to *select_singval_cmp()*, *select_singval_cmp2()*, *select_singval_cmp3()*, *select_singval_cmp4()*, *bd_singval()*, *bd_singval2()* or if *bd_lasq1()* or *bd_dqds()* are directly used to compute the singular values of BD by the DQDS algorithm).

See, for example, the description of *select_singval_cmp()* or *select_singval_cmp3()* for more details.

Synopsis:

```
call bd_deflate2( mat(:m,:n) , tauq(:min(m,n)) , taup(:min(m,n)) , d(:min(m,
↪n)) , e(:min(m,n)) , s(:p) , leftvec(:m,:p) , rightvec(:n,:p) , failure ,␣
↪max_qr_steps=max_qr_steps , ortho=ortho , scaling=scaling , inviter=inviter␣
↪)

call bd_deflate2( mat(:m,:n) , p(:n,:n) , d(:n) , e(:n) , s(:p) , leftvec(:m,
↪:p) , rightvec(:n,:p) , failure , max_qr_steps=max_qr_steps , ortho=ortho ,␣
↪scaling=scaling , inviter=inviter )

call bd_deflate2( mat(:m,:n) , tauq(:min(m,n)) , taup(:min(m,n)) ,␣
↪rlmat(:min(m,n),:min(m,n)) , d(:min(m,n)) , e(:min(m,n)) , s(:p) ,␣
↪leftvec(:m,:p) , rightvec(:n,:p) , failure , tauo=tauo(:min(m,n)) , max_
↪qr_steps=max_qr_steps , ortho=ortho , scaling=scaling , inviter=inviter )

call bd_deflate2( mat(:m,:n) , rlmat(:min(m,n),:min(m,n)) , p(:n,:n) ,
↪ d(:n) , e(:n) , s(:p) , leftvec(:m,:p) , rightvec(:n,:p) , failure␣
↪, tauo=tauo(:min(m,n)) ,  max_qr_steps=max_qr_steps , ortho=ortho ,␣
↪scaling=scaling , inviter=inviter )
```

Examples:

ex1_bd_deflate2.F90

ex1_bd_deflate2_bis.F90

ex1_bd_deflate2_ter.F90

ex2_bd_deflate2.F90

ex2_select_singval_cmp.F90

ex2_select_singval_cmp2.F90

ex2_select_singval_cmp3.F90

ex2_select_singval_cmp3_bis.F90

ex2_select_singval_cmp4.F90

ex2_select_singval_cmp4_bis.F90

product_svd_cmp()

Purpose:

product_svd_cmp() computes the singular value decomposition of the product of a m-by-n matrix A by the transpose of a p-by-n matrix B:

$$A * B^T = U * S * V^T$$

where A and B have more rows than columns (n<=min(m,p)), S is an n-by-n matrix which is zero except for its diagonal elements, U is a m-by-n orthogonal matrix, and V is a p-by-n orthogonal matrix. The diagonal elements of S are the singular values of A * B^T; they are real and non-negative. The columns of U and V are the left and right singular vectors of A * B^T, respectively.

Synopsis:

```
call product_svd_cmp( a(:m,:n) , b(:p,:n) , s(:n) , failure , sort=sort␣
↪, maxiter=maxiter , max_francis_steps=max_francis_step , perfect_
↪shift=perfect_shift , bisect=bisect , dqds=dqds )
```

ginv()

Purpose:

ginv() returns the generalized (e.g., Moore-Penrose) inverse MAT^+ of a m-by-n real matrix, MAT. The generalized inverse of MAT is a n-by-m matrix and is computed with the help of the SVD of MAT [Golub_VanLoan:1996].

Synopsis:

```
matginv(:n,:m) = ginv( mat(:m,:n) , tol=tol , maxiter=maxiter , max_francis_
↪steps=max_francis_step , perfect_shift=perfect_shift , bisect=bisect ,␣
↪dqds=dqds )
```

Examples:

ex1_ginv.F90

comp_ginv()

Purpose:

comp_ginv() computes the generalized (e.g., Moore-Penrose) inverse MAT^+ of a m-by-n real matrix, MAT. The generalized inverse of MAT is a n-by-m matrix and is computed with the help of the SVD of MAT [Golub_VanLoan:1996].

Synopsis:

```
call comp_ginv( mat(:m,:n) , failure, matginv(:n,:m), tol=tol ,␣
↪singvalues=singvalues(:min(m,n)) , krank=krank , mul_size=mul_size␣
↪, maxiter=maxiter , max_francis_steps=max_francis_step , perfect_
↪shift=perfect_shift , bisect=bisect , dqds=dqds )
call comp_ginv( mat(:m,:n) , failure ,                tol=tol ,␣
↪singvalues=singvalues(:min(m,n)) , krank=krank , mul_size=mul_size␣
↪, maxiter=maxiter , max_francis_steps=max_francis_step , perfect_
↪shift=perfect_shift , bisect=bisect , dqds=dqds )
```

Examples:

ex1_comp_ginv.F90

gen_bd_mat()

Purpose:

gen_bd_mat() generates different types of bidiagonal matrices with known singular values or specific numerical properties, such as clustered singular values or a large condition number, for testing purposes of SVD (bidiagonal) solvers included in STATPACK.

Optionally, the singular values of the selected bidiagonal matrix can be computed analytically, if possible, or by a bisection algorithm with high absolute and relative accuracies.

Synopsis:

```
call gen_bd_mat( type , d(:n) , e(:n) , failure=failure , known_singval=known_
↪singval , from_tridiag=from_tridiag , singval=singval(:n) , sort=sort ,␣
↪val1=val1 , val2=val2 , l0=l0 , glu0=glu0 )
```

5.21 MODULE LLSQ_Procedures

Module *LLSQ_Procedures* exports routines for solving linear least squares problems and related computations [Golub_VanLoan:1996] [Lawson_Hanson:1974] [Hansen_etal:2012].

More precisely, routines provided in the *LLSQ_Procedures* module compute solution of the problem

$$\min_x ||b - MAT * x||_2$$

where MAT is a m-by-n real matrix, b is a m-element vector and x a n-element vector and $||||_2$ is the 2-norm, or of the problem

$$\min_X ||B - MAT * X||_F$$

where MAT, B and X are m-by-n, m-by-p and n-by-p real matrices, respectively, and $||||_F$ is the Frobenius norm.

These linear least squares solvers are based on the QR decomposition, the QR decomposition with Column Pivoting (QRCP), the Complete Orthogonal Decomposition (COD) and the Singular Value Decomposition (SVD) provided by the *QR_Procedures* and *SVD_Procedures* modules. In addition, randomized versions of the QRCP and COD [Duersch_Gu:2017] [Martinsson_etal:2017] [Duersch_Gu:2020] [Martinsson:2019], which are available in module *Random* and are much faster than their deterministic counterparts, can also be used to solve the above two problems.

Assuming for simplicity that m>=n and MAT has full column rank, the QR decomposition of MAT is

$$MAT = Q * R$$

and the solution of the (first) linear least squares problem in that case is

$$x = R^{-1} * [Q^T * b](:n)$$

Assuming now that m>=n, but MAT has deficient column rank with $r = \text{rank}(\texttt{MAT})$, the QRCP of MAT is

$$MAT * P \simeq Q * \begin{bmatrix} R11 & R12 \\ 0 & 0 \end{bmatrix}$$

where P is a permutation of the columns of $\texttt{I}_\texttt{n}$, the identity matrix of order n, R11 is a r-by-r full rank upper triangular matrix and R12 is a r-by-(n-r) matrix. Using this QRCP decomposition, the so-called *basic* solution of the (first) linear least squares problem is now

$$x = P * \begin{pmatrix} R11^{-1} * [Q^T * b](:r) \\ 0 \end{pmatrix}$$

This solution has at least n-r zero components, which corresponds to using only the first r columns of MAT*P in the solution. Interestingly, this implies that the vector b can be *approximated* by the smallest subset of r columns of MAT.

Note, however, that in this case the solution of the linear least squares problem is not unique [Lawson_Hanson:1974] [Golub_VanLoan:1996] [Hansen_etal:2012] and it can be demonstrated that the general solution is now given by

$$x^* = P * \begin{pmatrix} R11^{-1} * ([Q^T * b](:r) - R12 * y) \\ y \end{pmatrix}$$

where `y` is an arbitrary `(n-r)`-element vector [Hansen_etal:2012].

Assuming now that `MAT` is a deficient matrix with $r = \text{rank}(MAT)$, the COD of `MAT` allows us to compute the unique minimum `2`-norm solution of our linear least squares problem as

$$x = P * Z^T \begin{pmatrix} T11^{-1} * [Q^T * b](:r) \\ 0 \end{pmatrix}$$

with the COD defined as

$$MAT * P = Q * \begin{bmatrix} T11 & 0 \\ 0 & 0 \end{bmatrix} * Z$$

where `T11` is a `r`-by-`r` upper triangular full rank matrix and `Z` is a `n`-by-`n` orthogonal matrix. See the manual of the *QR_Procedures* module for more details.

Finally, the SVD of `MAT` (which is a special case of the COD described above in which `P` is the identity matrix and `T11` is a diagonal matrix) allows to compute the generalized inverse of `MAT`, MAT^+, by setting to zero the smallest singular values of `MAT`, which are below a suitable threshold selected to estimate accurately the rank of `MAT` [Lawson_Hanson:1974] [Golub_VanLoan:1996] [Hansen_etal:2012]. Using such generalized inverse, the minimum `2`-norm solution of our linear least squares problem is given by

$$x = MAT^+ * b$$

See the *`llsq_svd_solve()`* or *`llsq_svd_solve2()`* subroutines for more details on using the SVD for solving linear least squares problems with maximum accuracy. These routines are particularly useful, when you suspect that `MAT` is rank-deficient or close to rank-deficient, as in these cases the solution `x` of the above linear least squares problem will be very sensitive to small perturbations in the data `MAT` and `b` if the rank of `MAT` is not properly estimated.

Please note that routines provided in this module apply only to real data of kind **stnd**. The real kind type **stnd** is defined in module *Select_Parameters*.

In order to use one of these routines, you must include an appropriate `use LLSQ_Procedures` or `use Statpack` statement in your Fortran program, like:

```
use LLSQ_Procedures, only: solve_llsq
```

or :

```
use Statpack, only: solve_llsq
```

Here is the list of the public routines exported by module *LLSQ_Procedures*:

`solve_llsq``()`

Purpose:

solve_llsq() computes a solution to a real linear least squares problem:

$$\min_X ||B - A * X||_2$$

using a QRCP or COD factorization of A. A is a m-by-n matrix which may be rank-deficient. m>=n or n>m is permitted.

B is a right hand side vector (or matrix), and X is a solution vector (or matrix).

The function returns the solution vector or matrix X. In case of a rank deficient matrix A, the minimum 2-norm solution can be computed at the user option (e.g., by using the optional logical argument *MIN_NORM* with the value true).

Input arguments *A* and *B* are not overwritten by **solve_llsq()**.

Synopsis:

```
x(:n)     = solve_llsq( a(:m,:n) , b(:m)     , krank=krank , tol=tol , min_
↪norm=min_norm )
x(:n,:nb) = solve_llsq( a(:m,:n) , b(:m,:nb) , krank=krank , tol=tol , min_
↪norm=min_norm )
x         = solve_llsq( a(:m)    , b(:m)     )
x(:nb)    = solve_llsq( a(:m)    , b(:m,:nb) )
```

Examples:

ex1_solve_llsq.F90

ex2_solve_llsq.F90

llsq_qr_solve()

Purpose:

llsq_qr_solve() computes a solution to a real linear least squares problem:

$$\min_X ||B - MAT * X||_2 \text{ or } \min_X ||B - VEC * X||_2$$

using a QRCP or COD factorization of MAT. Here MAT is a m-by-n matrix, which may be rank-deficient, m>=n or n>m is permitted, and VEC is a m-vector.

B is a right hand side vector (or matrix), and X is a solution vector (or matrix).

This subroutine computes the solution vector or matrix X as function *solve_llsq()* described above and, optionally, the rank of MAT, the residual vector (or matrix) of the linear least squares problem or its 2-norm. In case of a rank deficient matrix MAT, the minimum 2-norm solution can be computed at the user option (e.g., by using the optional logical argument *MIN_NORM* with the value true).

Input arguments *MAT*, *VEC* and *B* are not overwritten by **llsq_qr_solve()**.

Synopsis:

```
call llsq_qr_solve( mat(:m,:n) , b(:m)     , x(:n)     , rnorm=rnorm      ,␣
↪resid=resid(:m)     , krank=krank , tol=tol , min_norm=min_norm )
call llsq_qr_solve( mat(:m,:n) , b(:m,:nb) , x(:n,:nb) , rnorm=rnorm(:nb) ,␣
↪resid=resid(:m,:nb) , krank=krank , tol=tol , min_norm=min_norm )
call llsq_qr_solve( vec(:m)    , b(:m)     , x          , rnorm=rnorm      ,␣
↪resid=resid(:m)     )
call llsq_qr_solve( vec(:m)    , b(:m,:nb) , x(:nb)     , rnorm=rnorm(:nb) ,␣
↪resid=resid(:m,:nb) )
```

Examples:

ex1_llsq_qr_solve.F90

ex2_llsq_qr_solve.F90

ex3_llsq_qr_solve.F90

llsq_qr_solve2()

Purpose:

llsq_qr_solve2() computes a solution to a real linear least squares problem:

$$\min_X ||B - MAT * X||_2 \text{ or } \min_X ||B - VEC * X||_2$$

using a QRCP or COD factorization of MAT. Here MAT is a m-by-n matrix, which may be rank-deficient, m>=n or n>m is permitted, and VEC is a m-vector.

B is a right hand side vector (or matrix) and X is a solution vector (or matrix).

The subroutine computes the solution vector or matrix X and, optionally, the rank of MAT, the residual vector of the linear least squares problem or its 2-norm. In case of a rank deficient matrix MAT, the minimum 2-norm solution can be computed at the user option (e.g., by using the optional logical argument *MIN_NORM* with the value true).

Arguments *MAT*, *VEC* and *B* are overwritten with information generated by **llsq_qr_solve2()**. This is the main difference between **llsq_qr_solve2()** and subroutine *llsq_qr_solve()* described above.

A second difference with *llsq_qr_solve()* is that the QRCP or COD decompositions of MAT can be saved in arguments *MAT*, *DIAGR*, *BETA*, *IP* and *TAU* on exit at the user option and reused later for solving linear least squares problems with other right hand side vectors or matrices.

Synopsis:

```
call llsq_qr_solve2( mat(:m,:n) , b(:m)      , x(:n)      , rnorm=rnorm
↪  , comp_resid=comp_resid , krank=krank , tol=tol , min_norm=min_norm ,
↪diagr=diagr(:min(m,n)) , beta=beta(:min(m,n)) , ip=ip(:n) , tau=tau(:min(m,
↪n)) )
call llsq_qr_solve2( mat(:m,:n) , b(:m,:nb) , x(:n,:nb) , rnorm=rnorm(:nb)
↪, comp_resid=comp_resid , krank=krank , tol=tol , min_norm=min_norm ,
↪diagr=diagr(:min(m,n)) , beta=beta(:min(m,n)) , ip=ip(:n) , tau=tau(:min(m,
↪n)) )
call llsq_qr_solve2( vec(:m)     , b(:m)      , x          , rnorm=rnorm      ,
↪comp_resid=comp_resid , diagr=diagr , beta=beta )
call llsq_qr_solve2( vec(:m)     , b(:m,:nb) , x(:nb)     , rnorm=rnorm(:nb) ,
↪comp_resid=comp_resid , diagr=diagr , beta=beta )
```

Examples:

ex1_llsq_qr_solve2.F90

ex2_llsq_qr_solve2.F90

ex3_llsq_qr_solve2.F90

qr_solve()

Purpose:

qr_solve() solves overdetermined or underdetermined real linear systems

$$MAT * X = B$$

with a m-by-n matrix MAT, using a QR factorization of MAT as computed by *qr_cmp()* in module *QR_Procedures*. m>=n or n>m is permitted, but it is assumed that MAT has full rank.

B is a right hand side vector (or matrix) and X is a solution vector (or matrix).

It is assumed that *qr_cmp()* has been used to compute the QR factorization of MAT before calling **qr_solve()**. The input arguments *MAT*, *DIAGR* and *BETA* of **qr_solve()** give the QR factorization of MAT and assume the same formats as used for the corresponding output arguments of *qr_cmp()*.

Synopsis:

```
call qr_solve( mat(:m,:n) , diagr(:min(m,n)) , beta(:min(m,n)) , b(:m)      ,␣
↪x(:n)      , rnorm=rnorm       , comp_resid=comp_resid )
call qr_solve( mat(:m,:n) , diagr(:min(m,n)) , beta(:min(m,n)) , b(:m,:nb) ,␣
↪x(:n,:nb) , rnorm=rnorm(:nb) , comp_resid=comp_resid )
```

Examples:

ex1_qr_solve.F90

ex2_qr_cmp.F90

qr_solve2()

Purpose:

qr_solve2() solves overdetermined or underdetermined real linear systems

$$MAT * X = B$$

with a m-by-n matrix MAT, using a QRCP or COD decomposition of MAT as computed by subroutines *qr_cmp2()* or *partial_qr_cmp()* in module *QR_Procedures* or subroutines *partial_rqr_cmp()* and *partial_rqr_cmp2()* in module *Random*. m>=n or n>m is permitted and MAT may be rank-deficient.

B is a right hand side vector (or matrix) and X is a solution vector (or matrix).

In case of a rank deficient matrix MAT and a COD of MAT is used in input of **qr_solve2()**, the minimum 2-norm solution is computed.

It is assumed that *qr_cmp2()*, *partial_qr_cmp()*, *partial_rqr_cmp()* or *partial_rqr_cmp2()* have been used to compute the QRCP or COD of MAT before calling **qr_solve2()**.

The input arguments *MAT*, *DIAGR*, *BETA*, *IP* and *TAU* give the QRCP or COD of MAT and assume the same formats as used for the corresponding output arguments of *qr_cmp2()*, *partial_qr_cmp()*, *partial_rqr_cmp()* or *partial_rqr_cmp2()*.

Synopsis:

```
call qr_solve2( mat(:m,:n) , diagr(:min(m,n)) , beta(:min(m,n)) , ip(:n) ,␣
↪krank , b(:m)      , x(:n)      , rnorm=rnorm       , comp_resid=comp_resid ,␣
↪tau=tau(:min(m,n)) )
call qr_solve2( mat(:m,:n) , diagr(:min(m,n)) , beta(:min(m,n)) , ip(:n) ,␣
↪krank , b(:m,:nb) , x(:n,:nb) , rnorm=rnorm(:nb) , comp_resid=comp_resid ,␣
↪tau=tau(:min(m,n)) )
```

Examples:

ex1_qr_solve2.F90

ex2_qr_cmp2.F90

ex3_qr_cmp2.F90

ex3_partial_qr_cmp.F90

ex3_partial_rqr_cmp.F90

ex3_partial_rqr_cmp2.F90

rqb_solve()

Purpose:

rqb_solve() computes approximate solutions to overdetermined or underdetermined real linear systems

$$MAT * X = (Q * B) * X = C$$

with a m-by-n matrix MAT, using a randomized and approximate QRCP or COD decompositions of MAT as computed by subroutine *rqb_cmp()* in module *Random*. m>=n or n>m is permitted and MAT may be rank-deficient.

Here C is a right hand side vector (or matrix), X is an approximate solution vector (or matrix), Q is a m-by-nqb matrix with orthonormal columns and B is a nqb-by-n upper trapezoidal matrix as computed by *rqb_cmp()* with logical argument *COMP_QR* equals to true or with optional arguments *IP* and/or *TAU* present.

It is assumed that *rqb_cmp()* has been used to compute the randomized and approximate QRCP or COD of MAT before calling **rqb_solve()**.

The input arguments *Q*, *B*, *IP* and *TAU* give the randomized and approximate QRCP or COD of MAT and assume the same formats as used for the corresponding output arguments of *rqb_cmp()*.

Synopsis:

```
call rqb_solve( q(:m,:nqb) , b(:nqb,:n) , c(:m) ,     x(:n) ,     ip=ip(:n) ,␣
↪tau=tau(:nqb) , comp_resid=comp_resid )
call rqb_solve( q(:m,:nqb) , b(:nqb,:n) , c(:m,:nc) , x(:n,:nc) , ip=ip(:n) ,␣
↪tau=tau(:nqb) , comp_resid=comp_resid )
```

Examples:

ex1_rqb_solve.F90

llsq_svd_solve()

Purpose:

llsq_svd_solve() computes the minimum 2-norm solution to a real linear least squares problem:

$$\min_X ||B - MAT * X||_2$$

using the SVD of MAT. MAT is a m-by-n matrix which may be rank-deficient. A cache-efficient blocked and parallel version of the classic Golub and Kahan Householder bidiagonalization [Golub_VanLoan:1996] [Reinsch_Richter:2023], which reduces the traffic on the data bus from four reads and two writes per column-row elimination of the bidiagonalization process to one read and one write [Howell_etal:2008], is used in the first phase of the SVD algorithm.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m-by-nrhs right hand side matrix B and the n-by-nrhs solution matrix X, respectively.

The practical rank of MAT, krank, is determined by treating as zero those singular values which are less than *TOL* times the largest singular value.

Synopsis:

```
call llsq_svd_solve( mat(:m,:n) , b(:m)     , failure , x(:n)     ,␣
↪singvalues=singvalues(:min(m,n)) , krank=krank , rnorm=rnorm      , tol=tol␣
↪,  mul_size=mul_size , maxiter=maxiter , max_francis_steps=max_francis_
↪steps , perfect_shift=perfect_shift , bisect=bisect , dqds=dqds )
call llsq_svd_solve( mat(:m,:n) , b(:m,:nb) , failure , x(:n,:nb) ,␣
↪singvalues=singvalues(:min(m,n)) , krank=krank , rnorm=rnorm(:nb) , tol=tol␣
↪,  mul_size=mul_size , maxiter=maxiter , max_francis_steps=max_francis_
↪steps , perfect_shift=perfect_shift , bisect=bisect , dqds=dqds )
```

Examples:

ex1_llsq_svd_solve.F90

ex2_llsq_svd_solve.F90

llsq_svd_solve2()

Purpose:

llsq_svd_solve2() computes the minimum 2-norm solution to a real linear least squares problem:

$$\min_X ||B - MAT * X||_2$$

using the SVD of MAT. MAT is a m-by-n with m>=:data:*n* matrix which may be rank-deficient. A blocked (e.g., "BLAS3") and parallel version of the one-sided Ralha-Barlow bidiagonal reduction algorithm [Ralha:2003] [Barlow_etal:2005] [Bosner_Barlow:2007] is used in the first phase of the SVD algorithm.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m-by-nrhs right hand side matrix B and the n-by-nrhs solution matrix X, respectively.

The practical rank of MAT, krank, is determined by treating as zero those singular values which are less than *TOL* times the largest singular value.

In serial or parallel modes (e.g., when OpenMP is used) with a reduced number of processors, **llsq_svd_solve2()** can be significantly faster than **llsq_svd_solve()**. However, in parallel mode, but with a large number of processors, **llsq_svd_solve()** is likely to be faster then **llsq_svd_solve2()**.

Synopsis:

```
call llsq_svd_solve2( mat(:m,:n) , b(:m)     , failure , x(:n)     ,␣
↪singvalues=singvalues(:min(m,n)) , krank=krank , rnorm=rnorm      , tol=tol␣
↪,  mul_size=mul_size , maxiter=maxiter , max_francis_steps=max_francis_
↪steps , perfect_shift=perfect_shift , bisect=bisect , dqds=dqds )
call llsq_svd_solve2( mat(:m,:n) , b(:m,:nb) , failure , x(:n,:nb) ,␣
↪singvalues=singvalues(:min(m,n)) , krank=krank , rnorm=rnorm(:nb) , tol=tol␣
↪,  mul_size=mul_size , maxiter=maxiter , max_francis_steps=max_francis_
↪steps , perfect_shift=perfect_shift , bisect=bisect , dqds=dqds )
```

Examples:

ex1_llsq_svd_solve2.F90

5.22 MODULE Lin_Procedures

Module *Lin_Procedures* exports subroutines and functions for the solution of systems of linear equations, computing a triangular factorization (e.g., LU, Cholesky), computing the inverse of a square matrix or its determinant.

Routines in this module are blocked and multi-threaded versions of the standard algorithm based on the LU and Cholesky decompositions [Golub_VanLoan:1996] [Higham:2009] [Higham:2011].

A general n-by-n square matrix, MAT, has an LU decomposition into upper and lower triangular matrices:

$$P * MAT = L * U$$

where P is a permutation matrix, L is unit lower triangular matrix and U is upper triangular matrix [Higham:2011]. This LU decomposition is also valid for singular matrices. For square full-rank matrices, this decomposition can be used to convert the linear system $MAT * x = b$ into a pair of full-rank triangular systems ($L * y = P * b$, U * x = y), which can be solved by forward and backward-substitution [Higham:2011].

A symmetric, positive semidefinite square matrix MAT has a Cholesky decomposition into a product of a lower triangular matrix L and its transpose L^T [Higham:2009]:

$$MAT = L * L^T$$

or into a product of an upper triangular matrix U and its transpose U^T:

$$MAT = U^T * U$$

A symmetric matrix MAT is positive semidefinite if the quadratic form $x^T * MAT * x$ is non-negative for all x. In other words, the Cholesky decomposition can only be carried out only when all the eigenvalues of the matrix are positive or null. This decomposition can be used to convert the linear system $MAT * x = b$ into a pair of triangular systems ($L * y = b$, $L^T * x = y$), which can be solved by forward and back-substitution if all the eigenvalues of the matrix are positive [Higham:2009].

Algorithms for solving linear squares systems and for computing the inverse or the determinant of a general or positive symmetric n-by-n square matrix are based on these LU and Cholesky factorizations and the associated triangular systems.

Finally, routines for the LU factorization of a n-by-n symmetric tridiagonal matrix T as

$$T = P * L * U$$

where P is a permutation matrix, L is a unit lower tridiagonal matrix with at most one non-zero sub-diagonal elements per column and U is an upper triangular matrix with at most two non-zero super-diagonal elements per column are also provided. The factorizations are obtained by Gaussian elimination with partial pivoting and implicit row scaling or with partial pivoting and row interchanges [Golub_VanLoan:1996] [Higham:2011].

If the n-by-n symmetric tridiagonal matrix T is no singular, associated linear systems can also be solved by subroutines provided in this module.

Please note that routines provided in this module apply only to real data of kind **stnd**. The real kind type **stnd** is defined in module *Select_Parameters*.

In order to use one of these routines, you must include an appropriate `use Lin_Procedures` or `use Statpack` statement in your Fortran program, like:

```
use Lin_Procedures, only: lu_cmp
```

or :

```
use Statpack, only: lu_cmp
```

Here is the list of the public routines exported by module *Lin_Procedures*:

lu_cmp()

Purpose:

lu_cmp() computes the LU decomposition with partial pivoting and implicit row scaling of a given n-by-n real matrix MAT

$$P * MAT = L * U$$

where P is a permutation matrix, L is a n-by-n unit lower triangular matrix and U is a n-by-n upper triangular matrix. P is a permutation matrix, stored in argument *IP*, such that

$$P = P(n) * ... * P(1)$$

with P(i) is the identity with row i and *IP(i)* interchanged.

Synopsis:

```
call lu_cmp( mat(:n,:n) , ip(:n) , d1 , d2=d2 , tol=tol , small=small )
```

Examples:

ex1_lu_cmp.F90

ex2_lu_cmp.F90

lu_cmp2()

Purpose:

lu_cmp2() computes the LU decomposition with partial pivoting and implicit row scaling of a given `n`-by-`n` real matrix `MAT`

$$P * MAT = L * U$$

where `P` is a permutation matrix, L is a `n`-by-`n` unit lower triangular matrix and `U` is a `n`-by-`n` upper triangular matrix. `P` is a permutation matrix, stored in argument *IP*, such that

$$P = P(n) * ... * P(1)$$

with `P(i)` is the identity with row `i` and *IP(i)* interchanged.

If *D2* is present, **lu_cmp2()** computes the determinant of `MAT` as

```
determinant(MAT) = scale( D1, D2 )
```

If *B* is present, **lu_cmp2()** solves the system of linear equations

$$MAT * X = B$$

using the LU factorization with scaled partial pivoting of `MAT`. Here `B` is a `n`-vector.

If *MATINV* is present, **lu_cmp2()** computes the inverse of `MAT`

$$MATINV = MAT^{-1}$$

Synopsis:

```
call lu_cmp2( mat(:n,:n) , ip(:n) , d1 , d2=d2 , b=b(:n) , matinv=matinv(:n,
↪:n) , tol=tol , small=small )
```

Examples:

ex1_lu_cmp2.F90

chol_cmp()

Purpose:

chol_cmp() computes the Cholesky factorization of a `n`-by-`n` real symmetric positive definite matrix `MAT`. The factorization has the form

$$MAT = U^T * U$$, if *UPPER* = `true` or is absent,

and

$$MAT = L * L^T$$, if *UPPER* = `false`,

where `U` is an upper triangular matrix and `L` is a lower triangular matrix.

Synopsis:

```
call chol_cmp( mat(:n,:n) , invdiag(:n) , d1 , d2=d2 , upper=upper , tol=tol )
```

Examples:

ex1_chol_cmp.F90

ex2_chol_cmp.F90

ex1_random_eig_pos.F90

chol_cmp2()

Purpose:

chol_cmp2() computes the Cholesky factorization of a `n`-by-`n` real symmetric positive definite matrix `MAT`. The factorization has the form

$MAT = U^T * U$, if *UPPER* = `true` or is absent,

and

$MAT = L * L^T$, if *UPPER* = `false`,

where `U` is an upper triangular matrix and `L` is a lower triangular matrix.

If *D2* is present, **chol_cmp2()** computes the determinant of `MAT` as

```
determinant(MAT) = scale( D1, D2 )
```

If *B* is present, **chol_cmp2()** solves the system of linear equations

$MAT * X = B$

using the Cholesky factorization of `MAT`. Here `B` is a `n`-vector.

If *MATINV* is present, **chol_cmp2()** computes the inverse of `MAT`

$MATINV = MAT^{-1}$

Synopsis:

```
call chol_cmp2( mat(:n,:n) , invdiag(:n) , d1 , d2=d2 , b=b(:n) ,␣
↪matinv=matinv(:n,:n) , upper=upper , fill=fill , tol=tol )
```

Examples:

ex1_chol_cmp2.F90

gchol_cmp()

Purpose:

gchol_cmp() computes the Cholesky factorization of a `n`-by-`n` real symmetric positive semidefinite matrix `MAT`. The factorization has the form

$MAT = U^T * U$, if *UPPER* = `true` or is absent,

and

$MAT = L * L^T$, if *UPPER* = `false`,

where `U` is an upper triangular matrix and `L` is a lower triangular matrix.

Synopsis:

```
call gchol_cmp( mat(:n,:n) , invdiag(:n) , krank , d1 , d2=d2 , upper=upper ,␣
↪tol=tol )
```

Examples:

ex1_gchol_cmp.F90

ex2_gchol_cmp.F90

gchol_cmp2()

Purpose:

gchol_cmp2() computes the Cholesky factorization of a `n`-by-`n` real symmetric positive semidefinite matrix `MAT`. The factorization has the form

$MAT = U^T * U$, if *UPPER* = `true` or is absent,

and

$MAT = L * L^T$, if *UPPER* = `false`,

where U is an upper triangular matrix and L is a lower triangular matrix.

If *D2* is present, **gchol_cmp2()** computes the determinant of MAT as

```
determinant(MAT) = scale( D1, D2 )
```

If *B* is present, **gchol_cmp2()** solves the system of linear equations

$$MAT * X = B$$

using the Cholesky factorization of MAT if B belongs to the range of MAT. Here B is a n-vector. If B does not belongs to the range of MAT, an approximate solution is computed as

$$X = MATINV * B$$

where MATINV is a (generalized) inverse of MAT.

If *MATINV* is present, **gchol_cmp2()** computes a (generalized) inverse of MAT.

Synopsis:

```
call gchol_cmp2( mat(:n,:n) , invdiag(:n) , krank , d1 , d2=d2 , b=b(:n) ,␣
↪matinv=matinv(:n,:n) , upper=upper , fill=fill , tol=tol )
```

Examples:

ex1_gchol_cmp2.F90

lu_solve()

Purpose:

lu_solve() solves a system of linear equations

$$MAT * X = B$$

where MAT is a n-by-n coefficient matrix and B is a n-vector or a n-by-m matrix, using the LU factorization with scaled partial pivoting of MAT,

$$P * MAT = L * U$$

as computed by *lu_cmp()* or *lu_cmp2()*.

Synopsis:

```
call lu_solve( mat(:n,:n) , ip(:n) , b(:n)     )
call lu_solve( mat(:n,:n) , ip(:n) , b(:n,:m) )
```

Examples:

ex1_lu_cmp.F90

ex2_lu_cmp.F90

lu_solve2()

Purpose:

lu_solve2() solves a system of linear equations

$$MAT * X = B$$

where MAT is a n-by-n coefficient matrix and B is a n-vector or a n-by-m matrix, using the LU factorization with scaled partial pivoting of MAT

$$P * MAT = L * U$$

as computed by *lu_cmp()* or *lu_cmp2()*.

Synopsis:

```
call lu_solve2( mat(:n,:n) , ip(:n) , b(:n)    )
call lu_solve2( mat(:n,:n) , ip(:n) , b(:n,:m) )
```

solve_lin()

Purpose:

solve_lin() solves a system of linear equations

$$MAT * X = B$$

with a n-by-n coefficient matrix MAT. B is a n-vector or a n-by-m matrix.

The function returns the solution vector or matrix X, if the matrix MAT is not singular.

Synopsis:

```
x(:n)    = solve_lin( mat(:n,:n) , b(:n)    , tol=tol )
x(:n,:m) = solve_lin( mat(:n,:n) , b(:n,:m) , tol=tol )
```

Examples:

ex1_solve_lin.F90

ex2_solve_lin.F90

lin_lu_solve()

Purpose:

lin_lu_solve() solves a system of linear equations

$$MAT * X = B$$

with a n-by-n coefficient matrix MAT. B is a n-vector or a n-by-m matrix.

The LU decomposition with partial pivoting and implicit row scaling of the matrix MAT

$$P * MAT = L * U$$

where P is a permutation matrix, L is a n-by-n unit lower triangular matrix and U is a n-by-n upper triangular matrix, is used to solve the linear system.

Synopsis:

```
call lin_lu_solve( mat(:n,:n) , b(:n)    , failure ,            tol=tol ,␣
↪small=small )
call lin_lu_solve( mat(:n,:n) , b(:n,:m) , failure ,            tol=tol ,␣
↪small=small )
call lin_lu_solve( mat(:n,:n) , b(:n)    , failure , x(:n)    , tol=tol ,␣
↪small=small )
call lin_lu_solve( mat(:n,:n) , b(:n,:m) , failure , x(:n,:m) , tol=tol ,␣
↪small=small )
```

Examples:

ex1_lin_lu_solve.F90

ex2_lin_lu_solve.F90

chol_solve()

Purpose:

chol_solve() solves a system of linear equations

$$MAT * X = B$$

where MAT is a n-by-n symmetric positive definite matrix and B is a n-vector or a n-by-m matrix, using the Cholesky factorization MAT,

$$MAT = U^T * U \text{ or } MAT = L * L^T$$

as computed by *chol_cmp()* or *gchol_cmp()*.

Synopsis:

```
call chol_solve( mat(:n,:n) , invdiag(:n) , b(:n)    , upper=upper )
call chol_solve( mat(:n,:n) , invdiag(:n) , b(:n,:m) , upper=upper )
```

Examples:

ex1_chol_cmp.F90

ex2_chol_cmp.F90

ex2_gchol_cmp.F90

triang_solve()

Purpose:

triang_solve() solves a triangular system of the form

$$MAT * X = B \text{ or } MAT^T * X = B$$

where MAT is a triangular matrix of order n, and B is a n-vector or a n-by-m matrix.

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

Synopsis:

```
call triang_solve( mat(:n,:n) , b(:n)    ,        upper=upper , trans=trans )
call triang_solve( mat(:n,:n) , b(:n,:m) ,        upper=upper , trans=trans )
call triang_solve( mat(:n,:n) , b(:n)    , scal , upper=upper , trans=trans )
call triang_solve( mat(:n,:n) , b(:n,:m) , scal , upper=upper , trans=trans )
```

Examples:

ex1_h1.F90

ex1_hous1.F90

comp_inv()

Purpose:

comp_inv() computes the inverse of a real square matrix MAT.

Synopsis:

```
call comp_inv( mat(:n,:n) , failure ,                 tol=tol )
call comp_inv( mat(:n,:n) , failure , matinv(:n,:n) , tol=tol )
```

Examples:

ex1_comp_inv.F90

ex2_comp_inv.F90

inv()

Purpose:

inv() computes the inverse of a real square matrix MAT,

$$MAT * INV(MAT) = I$$

Synopsis:

```
matinv(:n,:n) = inv( mat(:n,:n) , tol=tol )
```

Examples:

ex1_inv.F90

comp_sym_inv()

Purpose:

comp_sym_inv() computes the inverse of a real symmetric positive definite matrix `MAT` using the Cholesky factorization of `MAT`:

$$MAT = U^T * U \text{ or } MAT = L * L^T$$

Synopsis:

```
call comp_sym_inv( mat(:n,:n) , failure ,                  upper=upper ,␣
↪fill=fill , tol=tol )
call comp_sym_inv( mat(:n,:n) , failure , matinv(:n,:n) , upper=upper ,␣
↪fill=fill , tol=tol )
```

Examples:

ex1_comp_sym_inv.F90

sym_inv()

Purpose:

sym_inv() computes the inverse of a real symmetric positive definite matrix `MAT` using the Cholesky factorization `MAT`:

$$MAT = U^T * U \text{ or } MAT = L * L^T$$

Synopsis:

```
matinv(:n,:n) = sym_inv( mat(:n,:n) , upper=upper , tol=tol )
```

Examples:

ex1_sym_inv.F90

comp_sym_ginv()

Purpose:

comp_sym_ginv() computes the (generalized) inverse of a real symmetric positive semidefinite matrix `MAT` using the Cholesky factorization `MAT`:

$$MAT = U^T * U \text{ or } MAT = L * L^T$$

Synopsis:

```
call comp_sym_ginv( mat(:n,:n) , failure , krank ,                 ␣
↪upper=upper , fill=fill , tol=tol )
call comp_sym_ginv( mat(:n,:n) , failure , krank , matinv(:n,:n) ,␣
↪upper=upper , fill=fill , tol=tol )
```

Examples:

ex1_comp_sym_ginv.F90

comp_triang_inv()

Purpose:

comp_triang_inv() computes the inverse of a real upper or lower triangular matrix `MAT`.

Synopsis:

```
call comp_triang_inv( mat(:n,:n) ,                  upper=upper )
call comp_triang_inv( mat(:n,:n) , matinv(:n,:n) , upper=upper )
```

Examples:

ex1_comp_triang_inv.F90

ex2_comp_triang_inv.F90

comp_uut_ltl()

Purpose:

comp_uut_ltl() computes the product

$U * U^T$ or $L^T * L$

where the triangular factor `U` or `L` is stored in the upper or lower triangular part of `MAT`.

Synopsis:

```
call comp_uut_ltl( mat(:n,:n) ,                upper=upper , fill=fill )
call comp_uut_ltl( mat(:n,:n) , prod(:n,:n) , upper=upper , fill=fill )
```

comp_det()

Purpose:

comp_det() computes the determinant of a real square matrix `MAT`

```
DET = determinant( MAT )
```

Synopsis:

```
call comp_det( mat(:n,:n) , det , tol=tol , man_det=man_det , exp_det=exp_det␣
↪)
```

Examples:

ex1_comp_det.F90

det()

Purpose:

det() computes the determinant of a real square matrix `MAT`

```
DET = determinant( MAT )
```

Synopsis:

```
matdet = det( mat(:n,:n) , tol=tol )
```

Examples:

ex1_det.F90

sym_trid_cmp()

Purpose:

sym_trid_cmp() factorizes an `n`-by-`n` symmetric tridiagonal matrix `T` as

$T = P * L * U$

where P is a permutation matrix, L is a unit lower tridiagonal matrix with at most one non-zero sub-diagonal elements per column and U is an upper triangular matrix with at most two non-zero super-diagonal elements per column.

The factorization is obtained by Gaussian elimination with partial pivoting and implicit row scaling.

Synopsis:

```
call sym_trid_cmp( d(:n), e(:n), sub(:n), diag(:n), sup1(:n), sup2(:n),␣
↪perm(:n), tol=tol )
```

sym_trid_cmp2()

Purpose:

sym_trid_cmp2() factorizes an n-by-n symmetric tridiagonal matrix T, as

$$T = P * L * U$$

where P is a permutation matrix, L is a unit lower tridiagonal matrix with at most one non-zero sub-diagonal elements per column and U is an upper triangular matrix with at most two non-zero super-diagonal elements per column.

The factorization is obtained by Gaussian elimination with partial pivoting and row interchanges.

Synopsis:

```
call sym_trid_cmp2( d(:n), e(:n), sub(:n), diag(:n), sup1(:n), sup2(:n),␣
↪perm(:n) )
```

sym_trid_solve()

Purpose:

sym_trid_solve() may be used to solve the system of linear equations

$$x * T = y$$

where T is an n-by-n symmetric tridiagonal matrix for x, following the factorization of T by *sym_trid_cmp()* or *sym_trid_cmp2()* as

$$T = P * L * U$$

where P is a permutation matrix, L is a unit lower tridiagonal matrix with at most one non-zero sub-diagonal elements per column and U is an upper triangular matrix with at most two non-zero super-diagonal elements per column.

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

Synopsis:

```
call sym_trid_solve( sub(:n), diag(:n), sup1(:n), sup2(:n), perm(:n), y(:n),␣
↪scale  )
call sym_trid_solve( sub(:n), diag(:n), sup1(:n), sup2(:n), perm(:n), y(:n)  ␣
↪      )
```

5.23 MODULE Prob_Procedures

Module *Prob_Procedures* exports subroutines and functions for probability distribution functions and their inverses.

A very good introduction to probability distribution functions and algorithms used in this module can be found in [Walck:2007].

In order to use one of these routines, you must include an appropriate `use Prob_Procedures` or `use Statpack` statement in your Fortran program, like:

```
use Prob_Procedures, only: lngamma
```

or:

```
use Statpack, only: lngamma
```

Here is the list of the public routines exported by module *Prob_Procedures*:

lngamma()

Purpose:

lngamma() evaluates the logarithm of the gamma function $\ln(\Gamma(x))$ for a strictly positive real argument *X*.

Argument *X* can be a scalar, a vector or a matrix.

The gamma function is defined as,

$$\Gamma(x) = \int_0^{+\infty} z^{x-1} e^{-z} dz$$

for `x > 0`.

This function uses a Lanczos-type approximation to $\ln(\Gamma(x))$ for `x > 0` [Lanczos:1964].

Its accuracy is about `14` significant digits except for small regions in the vicinity of `1` and `2`.

The function is parallelized when *X* is a vector or matrix argument, if OpenMP is used.

Synopsis:

```
lngam         = lngamma( x         )
lngam(:n)     = lngamma( x(:n)     )
lngam(:n,:m) = lngamma( x(:n,:m) )
```

probgamma()

Purpose:

probgamma() evaluates the gamma probability distribution function (e.g. the Incomplete Gamma Integral) for a positive real scalar (vector or matrix) argument *X* and a strictly positive value (vector or matrix) argument *GAMP* of the parameter `p` of the Gamma distribution.

probgamma() computes the probability that a random variable having a Gamma distribution with parameter `p` (given on input by *GAMP*) will be less than or equal to `x`:

$$probgamma = \frac{1}{\Gamma(p)} \int_0^x z^{p-1} e^{-z} dz = G(x, p)$$

For large *GAMP* (e.g. *GAMP* > `1000`), this function uses a normal approximation, based on the Wilson-Hilferty transformation, see [Abramowitz_Stegun:1970], Formula 26.4.14, for more details.

Otherwise, a Pearson's series expansion is used for evaluating the integral, see [Abramowitz_Stegun:1970], Formula 6.5.29. The *integrating* process is terminated when both the absolute and relative contributions to the integral is not greater than the value of the optional argument *ACU*. The default value for *ACU* gives the maximum precision of this function.

The time taken by this function thus depends in the precision requested through the *ACU* argument, and also varies slightly with the input arguments *X* and *GAMP*.

The function is parallelized when *X* is a vector or matrix argument, if OpenMP is used.

This function is more accurate than *probgamma3()*, but it may be slower.

Fore more details and algorithms, see [Lau:1980] and [Shea:1988].

Synopsis:

```
p        = probgamma( x        , gamp        , acu=acu , maxiter=maxiter ,␣
↪failure=failure )
p(:n)    = probgamma( x(:n)    , gamp        , acu=acu , maxiter=maxiter ,␣
↪failure=failure )
p(:n,:m) = probgamma( x(:n,:m) , gamp        , acu=acu , maxiter=maxiter ,␣
↪failure=failure )
p(:n)    = probgamma( x(:n)    , gamp(:n)    , acu=acu , maxiter=maxiter ,␣
↪failure=failure )
p(:n,:m) = probgamma( x(:n,:m) , gamp(:n,:m) , acu=acu , maxiter=maxiter ,␣
↪failure=failure )
```

probgamma2()

Purpose:

probgamma2() evaluates the gamma probability distribution function (e.g. the Incomplete Gamma Integral) for a positive real scalar (vector or matrix) argument *X* and a strictly positive value (vector or matrix) argument *GAMP* of the parameter p of the Gamma distribution.

probgamma2() computes the probability that a random variable having a Gamma distribution with parameter p (given on input by *GAMP*) will be less than or equal to x:

$$probgamma2 = \frac{1}{\Gamma(p)} \int_0^x z^{p-1} e^{-z} dz = G(x, p)$$

For large *GAMP* (e.g. *GAMP* > 1000), this function uses a normal approximation, based on the Wilson-Hilferty transformation, see [Abramowitz_Stegun:1970], Formula 26.4.14, for more details.

For *X* <= 1 or *X* < *GAMP*, a Pearson's series expansion is used, see [Abramowitz_Stegun:1970], Formula 6.5.29, p.262. For other values of *X*, a continued fraction expansion is used since this expansion tends to converge more quickly than Pearson's series expansion (used in *probgamma()*), see [Abramowitz_Stegun:1970], Formula 6.5.31, p.263.

In both cases, the *integrating* process is terminated when both the absolute and relative contributions to the integral is not greater than the value of *ACU*. The default value for *ACU* gives the maximum precision of this function.

The function is parallelized when *X* is a vector or matrix argument, if OpenMP is used.

The time taken by this function thus depends in the precision requested through the *ACU* argument, and also varies slightly with the input arguments *X* and *GAMP*.

This function is more accurate than *probgamma3()*, but it is slower.

Fore more details and algorithms, see [Lau:1980] and [Shea:1988].

Synopsis:

```
p        = probgamma2( x        , gamp        , acu=acu , maxiter=maxiter ,␣
↪failure=failure )
p(:n)    = probgamma2( x(:n)    , gamp        , acu=acu , maxiter=maxiter ,␣
↪failure=failure )
p(:n,:m) = probgamma2( x(:n,:m) , gamp        , acu=acu , maxiter=maxiter ,␣
↪failure=failure )
p(:n)    = probgamma2( x(:n)    , gamp(:n)    , acu=acu , maxiter=maxiter ,␣
↪failure=failure )
p(:n,:m) = probgamma2( x(:n,:m) , gamp(:n,:m) , acu=acu , maxiter=maxiter ,␣
↪failure=failure )
```

probgamma3()

Purpose:

probgamma3() evaluates the gamma probability distribution function (e.g. the Incomplete Gamma Integral) for a positive real scalar (vector or matrix) argument *X* and a strictly positive value (vector or matrix) argument *GAMP* of the parameter p of the Gamma distribution.

probgamma3() computes the probability that a random variable having a Gamma distribution with parameter p (given on input by *GAMP*) will be less than or equal to x:

$$probgamma3 = \frac{1}{\Gamma(p)} \int_0^x z^{p-1} e^{-z} dz = G(x, p)$$

For large *GAMP* (e.g. *GAMP* > 1000), this function uses a normal approximation, based on the Wilson-Hilferty transformation, see [Abramowitz_Stegun:1970], Formula 26.4.14, for more details.

For X<=max(GAMP/2,13), a Pearson's series expansion is used, see [Abramowitz_Stegun:1970], Formula 6.5.29, p.262. For larger values of *X*, an alternate Pearson's asymptotic series expansion is used since this expansion tends to converge more quickly [Shea:1988], see [Abramowitz_Stegun:1970], Formula 6.5.32, p.263.

In both cases, the *integrating* process is terminated when both the absolute and relative contributions to the integral is not greater than the value of *ACU*. The default value for *ACU* gives the maximum precision of this function.

The time taken by this function thus depends in the precision requested through the *ACU* argument, and also varies slightly with the input arguments *X* and *GAMP*.

The function is parallelized when *X* is a vector or matrix argument, if OpenMP is used.

probgamma3() is faster, but less accurate than *probgamma()* or *probgamma2()* since, for large values of *X*, the alternate Pearson's series expansion is only asymptotic.

Fore more details and algorithms, see [Lau:1980] and [Shea:1988].

Synopsis:

```
p        = probgamma3( x        , gamp        , acu=acu , maxiter=maxiter ,␣
↪failure=failure )
p(:n)    = probgamma3( x(:n)    , gamp        , acu=acu , maxiter=maxiter ,␣
↪failure=failure )
p(:n,:m) = probgamma3( x(:n,:m) , gamp        , acu=acu , maxiter=maxiter ,␣
↪failure=failure )
p(:n)    = probgamma3( x(:n)    , gamp(:n)    , acu=acu , maxiter=maxiter ,␣
↪failure=failure )
p(:n,:m) = probgamma3( x(:n,:m) , gamp(:n,:m) , acu=acu , maxiter=maxiter ,␣
↪failure=failure )
```

pinvgamma()

Purpose:

pinvgamma() evaluates the inverse gamma probability distribution function.

For given arguments *P* (0 <= *P* <= 1) and *GAMP* (*GAMP* > 0), *PINVGAMMA* returns the value x_p such that *P* is the probability that a random variable distributed as a gamma distribution with parameter gamp (given on input by *GAMP*) is less than or equal to x_p.

In other words, **pinvgamma()** returns the gamma deviate x_p corresponding to a given lower tail area of p of the gamma distribution with parameter gamp:

$$p = \frac{1}{\Gamma(gamp)} \int_0^{x_p} z^{gamp-1} e^{-z} dz = G(x_p, gamp)$$

This function actually uses the *pinvq2()* function and is adapted from [Best_Roberts:1975] [Shea:1988] [Shea:1991].

Synopsis:

```
x = pinvgamma( p , gamp , acu=acu , maxiter=maxiter )
```

probbeta()

Purpose:

probbeta() evaluates the beta probability distribution function (e.g the Incomplete Beta Function).

For given arguments *X* (`0` <= *X* <= `1`), *A* (*A* > `0`), *B* (*B* > `0`), **probbeta()** returns the probability that a random variable from a beta distribution having parameters `a` and `b` will be less than or equal to `x`,

$$probbeta = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \int_0^x z^{a-1}(1-z)^{b-1}dz$$

Argument *X* can be a scalar, a vector or a matrix.

The function is parallelized when *X* is a vector or matrix argument, if OpenMP is used.

This function is adapted from [Majumder_Bhattacharjee:1973] [Cran_etal:1977].

Synopsis:

```
p        = probbeta( x        , a , b , beta=beta , acu=acu , maxiter=maxiter␣
↪, failure=failure )
p(:n)    = probbeta( x(:n)    , a , b , beta=beta , acu=acu , maxiter=maxiter␣
↪, failure=failure )
p(:n,:m) = probbeta( x(:n,:m) , a , b , beta=beta , acu=acu , maxiter=maxiter␣
↪, failure=failure )
```

Examples:

ex1_probbeta.F90

pinvbeta()

Purpose:

pinvbeta() evaluates the inverse beta probability distribution function (e.g. the Incomplete Beta Function).

For given arguments *P* (`0` <= *P* <= `1`), *A* (*A* > `0.1`), *B* (*B* > `0.1`), **pinvbeta()** returns the value x_p such that `p` is the probability that a random variable distributed as $Beta(a, b)$ (e.g. the standard probability Beta distribution) is less than or equal to x_p.

In other words, **pinvbeta()** returns the beta deviate x_p corresponding to a given lower tail area of `p` of the beta distribution with parameters `a` and `a` (given on input by the arguments *A* and *B*, respectively):

$$p = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \int_0^{x_p} z^{a-1}(1-z)^{b-1}dz$$

This function is not very accurate for small values of *A* and/or *B* (e.g. less than `0.5`).

For more details and algorithms, see [Majumder_Bhattacharjee:1973] [Cran_etal:1977] [Berry_etal:1990] [Berry_etal:1991].

Synopsis:

```
x = pinvbeta( p , a , b , beta=beta , acu=acu , maxiter=maxiter )
```

probn()

Purpose:

probn() evaluates the standard normal (Gaussian) distribution function from *X* to infinity if *UPPER* is `true` or from minus infinity to *X* if UPPER is `false`.

In other words:

- if *UPPER* = `true` :

$$probn = prob(U > x) = \frac{1}{\sqrt{2\pi}} \int_x^{+\infty} \exp(-z^2/2)dz$$

- if *UPPER* = `false` :

$$probn = prob(U < x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp(-z^2/2)dz = \Phi(x)$$

for U following a standard normal distribution: $U \sim \mathcal{N}(0,\, 1)$.

It is accurate at least to 10 places (for double-precision data).

Real argument *X* is of kind **stnd** and the result of **probn()** is also returned as real data of kind **stnd**.

Argument *X* can be a scalar, a vector or a matrix.

The function is parallelized when *X* is a vector or matrix argument, if OpenMP is used.

This function is adapted from [Hill:1973].

Synopsis:

```
p         = probn( x         , upper )
p(:n)     = probn( x(:n)     , upper )
p(:n,:m)  = probn( x(:n,:m)  , upper )
```

Examples:

ex1_probn.F90

pinvn()

Purpose:

pinvn() evaluates the inverse of the standard normal (Gaussian) distribution function for the argument *P*, with 0 < *P* < 1,

$$x_p = \Phi(p)^{-1}$$

where $p = prob(U < x_p) = \Phi(x_p)$ for U following a standard normal distribution: $U \sim \mathcal{N}(0,\, 1)$.

In other words, **pinvn()** returns the normal deviate x_p corresponding to a given lower tail area of p of the standard normal distribution:

$$p = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x_p} \exp(-z^2/2)dz = \Phi(x_p)$$

The inverse Gaussian Cumulative Distribution Function (CDF) is approximated to high precision using rational approximations (polynomials with degree 2 and 3) by the subroutine PPND7 given in [Wichura:1988].

This function is accurate to about seven decimal figures for $min(p, 1-p) > 10^{-316}$.

Real argument *P* is of kind **stnd** and the result of **pinvn()** is also returned as real data of kind **stnd**.

Argument *P* can be a scalar, a vector or a matrix.

The function is parallelized when *P* is a vector or matrix argument, if OpenMP is used.

Synopsis:

```
x         = pinvn( p         )
x(:n)     = pinvn( p(:n)     )
x(:n,:m)  = pinvn( p(:n,:m)  )
```

Examples:

ex1_probn.F90

probn2()

Purpose:

probn2() evaluates the standard normal (Gaussian) distribution function from *X* to infinity if *UPPER* is true or from minus infinity to *X* if UPPER is false.

In other words:

- if *UPPER* = `true` :

$$probn = prob(U > x) = \frac{1}{\sqrt{2\pi}} \int_x^{+\infty} \exp(-z^2/2)dz$$

- if *UPPER* = `false` :

$$probn = prob(U < x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp(-z^2/2)dz = \Phi(x)$$

for `U` following a standard normal distribution: $U \sim \mathcal{N}(0,\, 1)$.

Real argument *X* is of kind **extd** and the result of **probn2()** is also returned as real data of kind **extd**.

Argument *X* can be a scalar, a vector or a matrix.

This function is parallelized when *X* is a vector or matrix argument if OpenMP is used.

probn2() is based upon algorithm 5666 for the error function from [Hart:1978] and is more accurate than *probn()*.

Synopsis:

```
p         = probn2( x         , upper )
p(:n)     = probn2( x(:n)     , upper )
p(:n,:m)  = probn2( x(:n,:m)  , upper )
```

Examples:

ex1_probn2.F90

pinvn2()

Purpose:

pinvn2() evaluates the inverse of the standard normal (Gaussian) distribution function for the argument *P*, with `0` < *P* < `1`,

$$x_p = \Phi(p)^{-1}$$

where $p = prob(U < x_p) = \Phi(x_p)$ for `U` following a standard normal distribution: $U \sim \mathcal{N}(0,\, 1)$.

In other words, **pinvn2()** returns the normal deviate x_p corresponding to a given lower tail area of `p` of the standard normal distribution:

$$p = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x_p} \exp(-z^2/2)dz = \Phi(x_p)$$

The inverse Gaussian Cumulative Distribution Function (CDF) is approximated to high precision using rational approximations (polynomials with degree 7) by the subroutine PPND16 given in [Wichura:1988].

This function is accurate to about seven decimal figures for $min(p, 1-p) > 10^{-316}$.

Real argument *P* is of kind **extd** and the result of **pinvn2()** is also returned as real data of kind **extd**.

Argument *P* can be a scalar, a vector or a matrix.

The function is parallelized when *P* is a vector or matrix argument, if OpenMP is used.

Synopsis:

```
x         = pinvn2( p         )
x(:n)     = pinvn2( p(:n)     )
x(:n,:m)  = pinvn2( p(:n,:m)  )
```

Examples:

ex1_probn2.F90

probt()

Purpose:

probt() evaluates the Student's t-distribution function with *NDF* degrees of freedom from *T* (*T* can be a scalar, a vector or a matrix) to infinity if *UPPER* is `true` or from minus infinity to *T* if *UPPER* is `false`.

In other words,

- if *UPPER* = `true` :

$$probt = prob(X > t) = \frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\Gamma(\nu/2)} \int_t^{+\infty} (1 + z^2/\nu)^{-(\nu+1)/2} dz$$

- if *UPPER* = `false`:

$$probt = prob(X < t) = \frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\Gamma(\nu/2)} \int_{-\infty}^{t} (1 + z^2/\nu)^{-(\nu+1)/2} dz$$

for `X` following a Student's t-distribution with ν degrees of freedom (given on input by the argument *NDF*): $X \sim t(\nu)$.

Argument *T* is of kind **stnd** and can be a scalar, a vector or a matrix.

This function is parallelized when *T* is a vector or matrix argument if OpenMP is used.

This function is adapted from [Cooper:1968] [Hill:1970].

Synopsis:

```
p         = probt( t         , ndf         , upper, ndf_max=ndf_max )
p(:n)     = probt( t(:n)     , ndf         , upper, ndf_max=ndf_max )
p(:n,:m)  = probt( t(:n,:m)  , ndf         , upper, ndf_max=ndf_max )
p(:n)     = probt( t(:n)     , ndf(:n)     , upper, ndf_max=ndf_max )
p(:n,:m)  = probt( t(:n,:m)  , ndf(:n,:m)  , upper, ndf_max=ndf_max )
```

Examples:

ex1_probt.F90

ex2_probt.F90

pinvt()

Purpose:

pinvt() evaluates the inverse of the Student's t distribution function with *NDF* degrees of freedom for the argument *P*, with `0` < *P* < `1` (*P* can be a scalar, a vector or a matrix).

In other words, **pinvt()** returns the quantile t_p of Student's t-distribution with ν degrees of freedom (given in the argument *NDF*) corresponding to a given lower tail area of `p`:

$$p = \frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\Gamma(\nu/2)} \int_{-\infty}^{t_p} (1 + z^2/\nu)^{-(\nu+1)/2} dz$$

Argument *P* is of kind **stnd** and can be a scalar, a vector or a matrix.

This function is parallelized when *P* is a vector or matrix argument if OpenMP is used.

This function is adapted from [Hill:1970b].

Synopsis:

```
t         = pinvt( p         , ndf         )
t(:n)     = pinvt( p(:n)     , ndf         )
t(:n,:m)  = pinvt( p(:n,:m)  , ndf         )
t(:n)     = pinvt( p(:n)     , ndf(:n)     )
t(:n,:m)  = pinvt( p(:n,:m)  , ndf(:n,:m)  )
```

Examples:

ex1_probt.F90

ex2_probt.F90

probstudent()

Purpose:

probstudent() evaluates the two-tailed probability of Student's t with *DF* degrees of freedom.

probstudent() computes the probability that a random variable following the Student's t distribution with ν degrees of freedom (given in the argument *DF*) will exceed `abs(t)` (*T* can be a scalar, a vector or a matrix) in absolute value:

$$probstudent = prob(abs(X) > abs(t)) = 2\frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\Gamma(\nu/2)}\int_{abs(t)}^{+\infty}(1+z^2/\nu)^{-(\nu+1)/2}dz$$

for `X` following a Student's t-distribution with ν degrees of freedom: $X \sim t(\nu)$.

This function is not very accurate for very small degrees of freedom (e.g. number of degrees of freedom less than `5`).

Argument *T* is of kind **stnd** and can be a scalar, a vector or a matrix.

This function is parallelized when *T* is a vector or matrix argument if OpenMP is used.

This function is adapted from [Hill:1970].

Synopsis:

```
p         = probstudent( t         , df         )
p(:n)     = probstudent( t(:n)     , df         )
p(:n,:m)  = probstudent( t(:n,:m)  , df         )
p(:n)     = probstudent( t(:n)     , df(:n)     )
p(:n,:m)  = probstudent( t(:n,:m)  , df(:n,:m)  )
```

Examples:

ex1_probstudent.F90

ex2_probstudent.F90

pinvstudent()

Purpose:

pinvstudent() evaluates the inverse of a modification of Student's t probability distribution function.

pinvstudent() calculates the two-tail quantile of Student's t-distribution with ν degrees of freedom (given in the argument *DF*), that is a positive value t_p such that the probability of the absolute value of `t` being greater than t_p is `p`,

$$p = 2\frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\Gamma(\nu/2)}\int_{t_p}^{+\infty}(1+z^2/\nu)^{-(\nu+1)/2}dz$$

Argument *P* is of kind **stnd** and can be a scalar, a vector or a matrix.

This function is parallelized when *P* is a vector or matrix argument if OpenMP is used.

Note that **pinvstudent()** does not provide the actual Student's t inverse. For `q` equals to the probability that a Student's t random variable is less than t_q (e.g. the *true* inverse of the Student's t distribution function), that inverse can be obtained with **pinvstudent()** by the following rules:

- for `q` in the range `[0.0,0.5]`, call **pinvstudent()** with `p = 2 * q` and negate the result t_p.
- for `q` in the range `[0.5,1.0]`, call **pinvstudent()** with `P = 2 * (1-q)`.

This function is adapted from [Hill:1970b].

Synopsis:

```
t         = pinvstudent( p         , df         )
t(:n)     = pinvstudent( p(:n)     , df         )
t(:n,:m)  = pinvstudent( p(:n,:m)  , df         )
```

```
t(:n)     = pinvstudent( p(:n)     , df(:n)     )
t(:n,:m)  = pinvstudent( p(:n,:m)  , df(:n,:m)  )
```

Examples:

ex1_probstudent.F90

ex2_probstudent.F90

ex1_probbeta.F90

probq()

Purpose:

probq() evaluates the chi-squared distribution function with *NDF* degrees of freedom from *X2* to infinity if *UPPER* is `true` or from zero to *X2* if *UPPER* is `false` for *X2* >= `0`.

In other words,

- if *UPPER* = `true` :

$$probq = prob(Q > x2) = \frac{1}{2\Gamma(\nu/2)} \int_{x2}^{+\infty} (z/2)^{\nu/2-1} \exp(-z/2) dz$$

- if *UPPER* = `false`:

$$probq = prob(Q < x2) = \frac{1}{2\Gamma(\nu/2)} \int_{0}^{x2} (z/2)^{\nu/2-1} \exp(-z/2) dz$$

for `Q` following the chi-squared distribution with ν degrees of freedom χ^2_ν (with ν given on input by the argument *NDF*): $Q \sim \chi^2_\nu$.

For *NDF* <= *NDF_MAX*, the chi-squared distribution function is integrating by using formulae 26.4.4 and 26.4.5 in [Abramowitz_Stegun:1970], otherwise a normal approximation based on the Wilson-Hilferty transformation is used (see [Abramowitz_Stegun:1970] Formula 26.4.14 and also [Wilson_Hilferty:1931]).

This function works for a scalar, vector or matrix argument *X2* and is parallelized when *X2* is a vector or matrix argument if OpenMP is used.

Note that **probq()** works only for integer degrees of freedom. It may be faster than *probq2()* or *probq3()* functions for the default value of *NDF_MAX*, but it is less accurate.

Synopsis:

```
p         = probq( x2         , ndf         , upper , ndf_max=ndf_max )
p(:n)     = probq( x2(:n)     , ndf         , upper , ndf_max=ndf_max )
p(:n,:m)  = probq( x2(:n,:m)  , ndf         , upper , ndf_max=ndf_max )
p(:n)     = probq( x2(:n)     , ndf(:n)     , upper , ndf_max=ndf_max )
p(:n,:m)  = probq( x2(:n,:m)  , ndf(:n,:m)  , upper , ndf_max=ndf_max )
```

Examples:

ex1_probq.F90

ex2_probq.F90

probq2()

Purpose:

probq2() evaluates the chi-squared distribution function with *DF* degrees of freedom from *X2* to infinity if *UPPER* is `true` or from zero to *X2* if *UPPER* is `false` for *X2* >= `0`.

In other words,

- if *UPPER* = `true` :

$$probq = prob(Q > x2) = \frac{1}{2\Gamma(\nu/2)} \int_{x2}^{+\infty} (z/2)^{\nu/2-1} \exp(-z/2) dz$$

- if *UPPER* = `false`:

$$probq = prob(Q < x2) = \frac{1}{2\Gamma(\nu/2)} \int_0^{x2} (z/2)^{\nu/2-1} \exp(-z/2) dz$$

for `Q` following the chi-squared distribution with ν degrees of freedom χ^2_ν (with ν given on input by the argument *DF*): $Q \sim \chi^2_\nu$.

If *DF* <= *DF_MAX*, the chi-squared distribution function is evaluated by integrating the incomplete Gamma integral (see [Abramowitz_Stegun:1970] formulae 6.5.29 and 6.5.32, and [Shea:1988] for more details), otherwise a normal approximation based on the Wilson-Hilferty transformation is used (see [Abramowitz_Stegun:1970] Formula 26.4.14, and also [Wilson_Hilferty:1931]).

This function works for a scalar, vector or matrix argument *X2* and is parallelized when *X2* is a vector or matrix argument if OpenMP is used.

Note that **probq2()** works for real degrees of freedom contrary to *probq()*. It is faster than *probq3()*, but it is less accurate.

Synopsis:

```
p        = probq2( x2        , df        , upper , df_max=df_max ,␣
↪maxiter=maxiter , failure=failure )
p(:n)    = probq2( x2(:n)    , df        , upper , df_max=df_max ,␣
↪maxiter=maxiter , failure=failure )
p(:n,:m) = probq2( x2(:n,:m) , df        , upper , df_max=df_max ,␣
↪maxiter=maxiter , failure=failure )
p(:n)    = probq2( x2(:n)    , df(:n)    , upper , df_max=df_max ,␣
↪maxiter=maxiter , failure=failure )
p(:n,:m) = probq2( x2(:n,:m) , df(:n,:m) , upper , df_max=df_max ,␣
↪maxiter=maxiter , failure=failure )
```

Examples:

ex1_probq2.F90

ex2_probq2.F90

probq3()

Purpose:

probq3() evaluates the chi-squared distribution function with *DF* degrees of freedom from *X2* to infinity if *UPPER* is `true` or from zero to *X2* if *UPPER* is `false` for *X2* >= `0`.

In other words,

- if *UPPER* = `true` :

$$probq = prob(Q > x2) = \frac{1}{2\Gamma(\nu/2)} \int_{x2}^{+\infty} (z/2)^{\nu/2-1} \exp(-z/2) dz$$

- if *UPPER* = `false`:

$$probq = prob(Q < x2) = \frac{1}{2\Gamma(\nu/2)} \int_0^{x2} (z/2)^{\nu/2-1} \exp(-z/2) dz$$

for `Q` following the chi-squared distribution with ν degrees of freedom χ^2_ν (with ν given on input by the argument *DF*): $Q \sim \chi^2_\nu$.

If *DF* <= *DF_MAX*, the chi-squared distribution function is evaluated by integrating the incomplete Gamma integral (see [Abramowitz_Stegun:1970] formulae 6.5.29 and 6.5.31, and [Shea:1988] for more details), otherwise a normal approximation based on the Wilson-Hilferty transformation is used (see [Abramowitz_Stegun:1970] Formula 26.4.14, and also [Wilson_Hilferty:1931]).

This function works for a scalar, vector or matrix argument *X2* and is parallelized when *X2* is a vector or matrix argument if OpenMP is used.

Note that **probq3()** works for real degrees of freedom contrary to *probq()*. It is slower than *probq()* or *probq2()*, but it is more accurate.

Synopsis:

```
p        = probq3( x2        , df        , upper , df_max=df_max , acu=acu ,␣
↪maxiter=maxiter , failure=failure )
p(:n)    = probq3( x2(:n)    , df        , upper , df_max=df_max , acu=acu ,␣
↪maxiter=maxiter , failure=failure )
p(:n,:m) = probq3( x2(:n,:m) , df        , upper , df_max=df_max , acu=acu ,␣
↪maxiter=maxiter , failure=failure )
p(:n)    = probq3( x2(:n)    , df(:n)    , upper , df_max=df_max , acu=acu ,␣
↪maxiter=maxiter , failure=failure )
p(:n,:m) = probq3( x2(:n,:m) , df(:n,:m) , upper , df_max=df_max , acu=acu ,␣
↪maxiter=maxiter , failure=failure )
```

Examples:

ex1_probq3.F90

pinvq()

Purpose:

pinvq() evaluates the inverse of the chi-squared distribution function with *NDF* degrees of freedom for the argument *P*, with `0` < *P* < `1` (*P* can be a scalar, a vector or a matrix). **pinvq()** returns the quantile $x2_p$ of the chi-squared distribution with ν degrees of freedom (given in the argument *NDF*) corresponding to a given lower tail area of `p`.

In other words, **pinvq()** outputs a chi-squared value $x2_p$ such that a random variable, distributed as chi-squared with ν degrees of freedom will be less than $x2_p$ with probability `p`.

$$p = \frac{1}{2\Gamma(\nu/2)} \int_0^{x2_p} (z/2)^{\nu/2-1} \exp(-z/2) dz$$

This function is parallelized when *P* is a vector or matrix argument if OpenMP is used.

pinvq() is fast, but not very accurate, especially for small degrees of freedom, e.g. for *NDF* < `10` or `20`. If high accuracy is desired, function *pinvq2()* must be used instead. Moreover, **pinvq()** works only for integer degrees of freedom *NDF*.

This function is adapted from [Goldstein:1973]

Synopsis:

```
x2        = pinvq( p        , ndf        )
x2(:n)    = pinvq( p(:n)    , ndf        )
x2(:n,:m) = pinvq( p(:n,:m) , ndf        )
x2(:n)    = pinvq( p(:n)    , ndf(:n)    )
x2(:n,:m) = pinvq( p(:n,:m) , ndf(:n,:m) )
```

Examples:

ex1_probq.F90

ex2_probq.F90

pinvq2()

Purpose:

pinvq2() evaluates the inverse of the chi-squared distribution function with *DF* degrees of freedom for the argument *P*, with `0` < *P* < `1` (*P* can be a scalar, a vector or a matrix). **pinvq2()** returns the quantile $x2_p$ of the chi-squared distribution with ν degrees of freedom (given in the argument *DF*) corresponding to a given lower tail area of `p`

In other words, **pinvq2()** outputs a chi-squared value $x2_p$ such that a random variable, distributed as chi-squared with ν degrees of freedom will be less than $x2_p$ with probability `p`.

$$p = \frac{1}{2\Gamma(\nu/2)} \int_0^{x2_p} (z/2)^{\nu/2-1} \exp(-z/2) dz$$

This function is parallelized when *P* is a vector or matrix argument if OpenMP is used.

pinvq2() is both more general (here the number of degrees of freedom, *DF*, is not necessarily an integer) and more accurate (here the quantile $x2_p$ may be calculated as exactly as the computer allows with the parameter *PREC*) than *pinvq()* function.

This function is adapted from [Best_Roberts:1975] and [Shea:1991]

Synopsis:

```
x2         = pinvq2( p         , df         , prec=prec , acu=acu ,␣
↪maxiter=maxiter )
x2(:n)     = pinvq2( p(:n)     , df         , prec=prec , acu=acu ,␣
↪maxiter=maxiter )
x2(:n,:m)  = pinvq2( p(:n,:m)  , df         , prec=prec , acu=acu ,␣
↪maxiter=maxiter )
x2(:n)     = pinvq2( p(:n)     , df(:n)     , prec=prec , acu=acu ,␣
↪maxiter=maxiter )
x2(:n,:m)  = pinvq2( p(:n,:m)  , df(:n,:m)  , prec=prec , acu=acu ,␣
↪maxiter=maxiter )
```

Examples:

ex1_probq2.F90

ex1_probq3.F90

ex2_probq2.F90

probf()

Purpose:

probf() evaluates the F-distribution function with degrees of freedom *NDF1* and *NDF2* from *F* to infinity if *UPPER* is `true` or from zero to *F* if *UPPER* is `false` for a given input argument *F* >= `0`.

If Y_1 and Y_2 are chi-squared deviates with ν_1 and ν_2 degrees of freedom, respectively, then the ratio,

$$X = \frac{(Y_1/\nu_1)}{(Y_2/\nu_2)}$$

has an F-distribution $F(x; \nu_1, \nu_2)$.

Thus [Walck:2007],

- if *UPPER* = `true` :

$$probf = prob(X > f) = \int_f^{+\infty} \frac{\Gamma((\nu_1+\nu_2)/2)}{\Gamma(\nu_1/2)\Gamma(\nu_2/2)} \nu_1^{\nu_1/2} \nu_2^{\nu_2/2} f^{\nu_1/2-1} (\nu_2 + \nu_1 x)^{-\nu_1/2-\nu_2/2} df$$

- if *UPPER* = `false`:

$$probf = prob(X < f) = \int_0^{f} \frac{\Gamma((\nu_1+\nu_2)/2)}{\Gamma(\nu_1/2)\Gamma(\nu_2/2)} \nu_1^{\nu_1/2} \nu_2^{\nu_2/2} f^{\nu_1/2-1} (\nu_2 + \nu_1 x)^{-\nu_1/2-\nu_2/2} df$$

where Γ is the usual Gamma function and `X` follows an F-distribution with ν_1 and ν_2 degrees of freedom (given on input by the real arguments *DF1* and *DF2*, respectively): $X \sim F(x; \nu_1, \nu_2)$.

Argument *F* can be a scalar, a vector or a matrix.

This function is parallelized when *F* is a vector or matrix argument if OpenMP is used.

probf() accepts only integer values of degree of freedom and uses a normal approximation. See formula 2.24a in [Peizer_Pratt:1968] for more details. This normal approximation is not accurate for small values of degrees of freedom.

Synopsis:

```
p         = probf( f         , ndf1         , ndf2         , upper )
p(:n)     = probf( f(:n)     , ndf1         , ndf2         , upper )
p(:n,:m)  = probf( f(:n,:m)  , ndf1         , ndf2         , upper )
p(:n)     = probf( f(:n)     , ndf1(:n)     , ndf2(:n)     , upper )
p(:n,:m)  = probf( f(:n,:m)  , ndf1(:n,:m)  , ndf2(:n,:m)  , upper )
```

probf2()

Purpose:

probf2() evaluates the F-distribution function with degrees of freedom *DF1* and *DF2* from *F* to infinity if *UPPER* is `true` or from zero to *F* if *UPPER* is `false` for a given input argument *F* greater than or equal to zero.

If Y_1 and Y_2 are chi-squared deviates with ν_1 and ν_2 degrees of freedom, respectively, then the ratio,

$$X = \frac{(Y_1/\nu_1)}{(Y_2/\nu_2)}$$

has an F-distribution $F(x;\nu_1,\nu_2)$.

Thus [Walck:2007],

- if *UPPER* = `true` :

$$probf2 = prob(X > f) = \int_f^{+\infty} \frac{\Gamma((\nu_1+\nu_2)/2)}{\Gamma(\nu_1/2)\Gamma(\nu_2/2)} \nu_1^{\nu_1/2} \nu_2^{\nu_2/2} f^{\nu_1/2-1} (\nu_2+\nu_1 x)^{-\nu_1/2-\nu_2/2} df$$

- if *UPPER* = `false`:

$$probf2 = prob(X < f) = \int_0^{f} \frac{\Gamma((\nu_1+\nu_2)/2)}{\Gamma(\nu_1/2)\Gamma(\nu_2/2)} \nu_1^{\nu_1/2} \nu_2^{\nu_2/2} f^{\nu_1/2-1} (\nu_2+\nu_1 x)^{-\nu_1/2-\nu_2/2} df$$

where Γ is the usual Gamma function and `X` follows an F-distribution with ν_1 and ν_2 degrees of freedom (given on input by the real arguments *DF1* and *DF2*, respectively): $X \sim F(x;\nu_1,\nu_2)$.

Argument *F* can be a scalar, a vector or a matrix.

This function is parallelized when *F* is a vector or matrix argument if OpenMP is used.

probf2() accepts real values of degree of freedom and, actually, invokes the Beta distribution function *probbeta()* for computing the probability associated with the F-distribution [Walck:2007]. Thus, **probf2()** is much more accurate than *probf()*, but it is slower.

Synopsis:

```
p         = probf2( f         , df1         , df2         , upper , beta=beta ,␣
↪acu=acu , maxiter=maxiter , failure=failure )
p(:n)     = probf2( f(:n)     , df1         , df2         , upper , beta=beta ,␣
↪acu=acu , maxiter=maxiter , failure=failure )
p(:n,:m)  = probf2( f(:n,:m)  , df1         , df2         , upper , beta=beta ,␣
↪acu=acu , maxiter=maxiter , failure=failure )
p(:n)     = probf2( f(:n)     , df1(:n)     , df2(:n)     , upper , beta=beta ,␣
↪acu=acu , maxiter=maxiter , failure=failure )
p(:n,:m)  = probf2( f(:n,:m)  , df1(:n,:m)  , df2(:n,:m)  , upper , beta=beta ,␣
↪acu=acu , maxiter=maxiter , failure=failure )
```

pinvf2()

Purpose:

pinvf2() evaluates the inverse F probability distribution function with degrees of freedom *DF1* and *DF2* (integer or fractional degrees of freedom > `0.2`), for the given argument *P* with `0` < *P* < `1`.

Thus, **pinvf2()** returns the quantile f_p of the F-distribution $F(x;\nu_1,\nu_2)$ (where ν_1 and ν_2 are given in *DF1* and *DF2*, respectively) corresponding to a given lower tail area of `p`

In other words, **pinvf2()** outputs a F value f_p such that a random variable, distributed as a F-distribution with ν_1 and ν_2 degrees of freedom will be less than f_p with probability `p`.

$$p = \int_0^{f_p} \frac{\Gamma((\nu_1+\nu_2)/2)}{\Gamma(\nu_1/2)\Gamma(\nu_2/2)} \nu_1^{\nu_1/2} \nu_2^{\nu_2/2} f^{\nu_1/2-1} (\nu_2 + \nu_1 x)^{-\nu_1/2-\nu_2/2} df$$

where Γ is the usual Gamma function.

This function actually invoked the inverse BETA distribution function *pinvbeta()* for computing the value f_p associated with the probability *P*.

P can be a scalar, a vector or a matrix and this function is parallelized when *P* is a vector or matrix argument if OpenMP is used.

This function is not very accurate for small values of *DF1* and/or *DF2* (e.g. less than `1`).

Synopsis:

```
f = pinvf2( p , df1 , df2 , beta=beta , acu=acu , maxiter=maxiter )
```

probbinom()

Purpose:

probbinom() evaluates the cumulative binomial probability distribution function for a positive real argument *PROB* (with `0` <= *PROB* <= `1`), a strictly positive integer *N* and a positive integer *K* less than or equal to *N*.

probbinom() computes the probability that an event occurring with probability *PROB* per trial, will occur:

- *K* or more times in *N* independent trials if *UPPER* is `true`:

  $$probbinom = \sum_{i=k}^{n} \frac{n!}{i!(n-i)!} prob^i (1-prob)^{n-i}$$

- *K* or less times in *N* independent trials if *UPPER* is `false`:

  $$probbinom = \sum_{i=0}^{k} \frac{n!}{i!(n-i)!} prob^i (1-prob)^{n-i}$$

This probability is estimated with the help of the incomplete Beta function, as computed by function *probbeta()*, and the optional arguments *BETA*, *ACU*, *MAXITER* and *FAILURE* are passed directly to *probbeta()* if these arguments are present.

Synopsis:

```
p = probbinom( prob , n , k , upper , beta=beta , acu=acu , maxiter=maxiter ,␣
↪failure=failure )
```

rangen()

Purpose:

rangen() evaluates the probability that the normal range (e.g. the standardized difference between the maximum and the minimum on a sample) will be less than *X* (*X* > `0`) for a normal sample of size *N*.

For algorithm and details, see [Barnard:1978]

Synopsis:

```
p      = rangen( x      , n )
p(:n) = rangen( x(:n) , n )
```

5.24 MODULE Stat_Procedures

Module *Stat_Procedures* exports routines for univariate statistical computations.

All the routines in the *Stat_Procedures* module compute the different univariate statistics with only one pass through the data and recurrence relationships to average quantities in a stable way. Moreover, the routines can also be used to compute intermediate estimates of mean and variance on different chunks of data (eventually by different OpenMP threads), which can be merged later. This leads to high performance and out-of-core parallel methods on huge datasets. The routines may also take care of missing values in the data.

The statistical univariate procedures in the *Stat_Procedures* module include routines to compute the mean, variance, standard deviation, skewness, kurtosis and median on a (multi-channel) sample [vonStorch_Zwiers:2002].

The arithmetic mean, or *sample mean*, is denoted by $\hat{\mu}$ and defined as,

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

where x_i are the observations in a sample with `n` observations. For samples drawn from a gaussian distribution the variance of $\hat{\mu}$ itself is σ^2/n where σ^2 is the variance in the parent population.

The estimated variance in a sample with `n` observations is denoted by $\hat{\sigma}^2$ and is defined by,

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu})^2$$

or

$$\hat{\sigma}^2 = \frac{1}{(n-1)}\sum_{i=1}^{n}(x_i - \hat{\mu})^2$$

where x_i are the elements of the sample. Note that the normalization factor of $1/(n-1)$ results from the derivation of $\hat{\sigma}^2$ as an unbiased estimator of the population variance σ^2. For samples drawn from a Gaussian distribution the variance of $\hat{\sigma}^2$ itself is $2\sigma^4/n$.

The standard deviation is just defined as the square root of the variance.

The estimated skewness computed on a sample with `n` observations, is defined as,

$$\widehat{skew} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{x_i - \hat{\mu}}{\hat{\sigma}}\right)^3$$

or

$$\widehat{skew} = \frac{n}{(n-1)(n-2)}\sum_{i=1}^{n}\left(\frac{x_i - \hat{\mu}}{\hat{\sigma}}\right)^3$$

where x_i are the elements of the sample and $\hat{\sigma}$ is the unbiased estimate of the standard-deviation computed on the sample. Note that the normalization factor of $n/((n-1)(n-2))$ in the second definition of $\widehat{skew}$ results from the derivation of $\widehat{skew}$ has an unbiased estimator of the population skewness $skew$. The first biased definition is the classical formulae used in most textbooks [vonStorch_Zwiers:2002]. The skewness measures the deviation of a distribution from symmetry. For a symmetrical distribution, the skewness coefficient is always equal to zero, but the

converse is not true. Skewness is zero for a normal distribution. For unimodal distributions shifted to the right (left), the skewness coefficient is positive (negative). The skewness is useful to diagnose nonlinear processes and deviation from linearity.

In order to interpret correctly the skewness computed on a sample, note that the Standard Error (SE) of the skewness coefficient (e.g., the standard-deviation of $\widehat{skew}$ around $skew$) calculated on a sample drawn from a Gaussian distribution is given by:

$$SE(\widehat{skew}) = \sqrt{\frac{6n(n-1)}{(n-2)(n+1)(n+3)}}$$

This SE is not very different from $\sqrt{6/n}$ when the number of observations `n` is sufficiently high.

Moreover, the quantity $\widehat{skew}/SE(\widehat{skew})$ follows asymptotically a Gaussian distribution with mean `0` and variance equal to `1` when the sample is drawn from a Gaussian distribution. With a sample of independent Gaussian observations, a value twice the SE is thus associated with a 5% significance level suggesting a significant departure from a Gaussian distribution when the number of observations is sufficiently large.

The estimated kurtosis, computed on a sample with `n` observations, is defined as,

$$\widehat{kurt} = \left(\frac{n(n+1)}{(n-1)(n-2)(n-3)}\sum_{i=1}^{n}\left(\frac{x_i-\hat{\mu}}{\hat{\sigma}}\right)^4\right) - \left(\frac{3(n-1)^2}{(n-2)(n-3)}\right)$$

or

$$\widehat{kurt} = \left(\frac{1}{n}\sum_{i=1}^{n}\left(\frac{x_i-\hat{\mu}}{\hat{\sigma}}\right)^4\right) - 3$$

The first definition is the unbiased estimator of the population kurtosis $kurt$ and the second is the classical biased (but simpler) formulae used in most statistical textbooks [vonStorch_Zwiers:2002].

The kurtosis measures the flatness or peakedness of a distribution, i.e. how sharply peaked a distribution is, relative to its width. The kurtosis, as defined above, is normalized to zero for a Gaussian distribution and is always greater or equal to `-2`. In most cases, if the kurtosis is greater (lower) than zero then the distribution is more peaked (flatter) than the normal distribution with the same mean and standard-deviation.

In order to interpret correctly the kurtosis computed on a sample, note that the SE of the kurtosis coefficient calculated on a sample drawn from a Gaussian distribution is given by:

$$SE(\widehat{kurt}) = \sqrt{\frac{24n(n-1)^2}{(n-3)(n-2)(n+3)(n+5)}}$$

and the quantity $\widehat{kurt}/SE(\widehat{kurt})$ follows asymptotically a Gaussian distribution with mean `0` and variance equal to `1` when the sample is drawn from a Gaussian distribution.

Extreme departures from the mean will cause very high (absolute) values of kurtosis. Consequently, the kurtosis coefficient can be used to detect extreme observations or outliers in a sample of observations.

In summary, if you are interested in how well a distribution can be approximated by the normal distribution, the skewness and kurtosis coefficients and their standard errors can give you some useful information.

Unbiased estimators of variance, standard-deviation, skewwness and kurtosis can be computed by the *comp_unistat()* and *comp_unistat_miss()* subroutines. Biased estimates of variance and standard-deviation are computed by the *comp_mvs()* and *comp_mvs_miss()* subroutines.

Finally, procedures for performing composite analysis (e.g., testing differences of means between groups of observations) are also provided in this module [Terray_etal:2003].

Please note that routines provided in this module apply only to real data of kind **stnd**. The real kind type **stnd** is defined in module *Select_Parameters*.

In order to use one of these routines, you must include an appropriate `use Stat_Procedures` or `use Statpack` statement in your Fortran program, like:

```
use Stat_Procedures, only: comp_unistat
```

or:

```
use Statpack, only: comp_unistat
```

Here is the list of the public routines exported by module *Stat_Procedures*:

comp_unistat()

Purpose:

comp_unistat() computes estimates of univariate statistics from a data array, *X*. *X* can be a vector, a matrix or a tri- or four-dimensional array of data.

The subroutine computes the univariate statistics with only one pass through the data.

If all the data are not available at once, **comp_unistat()** can operate on chunks of data.

On output, the argument *XSTAT* will contain the following statistics:

- *XSTAT(...,1)* contains the mean value of the data vector.
- *XSTAT(...,2)* contains the variance of the data vector.
- *XSTAT(...,3)* contains the standard deviation of the data vector.
- *XSTAT(...,4)* contains the coefficient of skewness of the data vector.
- *XSTAT(...,5)* contains the coefficient of kurtosis of the data vector.
- *XSTAT(...,6)* contains the minimum of the data vector.
- *XSTAT(...,7)* contains the maximum of the data vector.

comp_unistat() computes unbiased estimates of variance and standard deviation. Unbiased estimates of skewness and kurtosis are computed only if the *NOBIAS* logical argument is used with the value `true`.

Synopsis:

```
call comp_unistat( x(:n)          , first , last ,          xstat(:7)       ,␣
↪xnobs=xnobs ,        nobias=nobias                 )
call comp_unistat( x(:m,:n)       , first , last ,          xstat(:m,:7)    ,␣
↪xnobs=xnobs ,        nobias=nobias , dimvar=dimvar )
call comp_unistat( x(:m,:p,:n)    , first , last ,          xstat(:m,:p,:7) ,␣
↪xnobs=xnobs ,        nobias=nobias                 )
call comp_unistat( x(:m,:p,:q,:n) , first , last ,          xstat(:m,:p,:7) ,␣
↪xnobs=xnobs ,        nobias=nobias                 )
call comp_unistat( x(:n)          , first , last , xmiss , xstat(:7)       ,␣
↪xnobs=xnobs ,        nobias=nobias                 )
```

```
call comp_unistat( x(:m,:n)       , first , last , xmiss , xstat(:m,:7)    ,␣
↪xnobs=xnobs(:m) ,    nobias=nobias , dimvar=dimvar )
call comp_unistat( x(:m,:p,:n)    , first , last , xmiss , xstat(:m,:p,:7) ,␣
↪xnobs=xnobs(:m,:p) , nobias=nobias                 )
call comp_unistat( x(:m,:p:q,,:n) , first , last , xmiss , xstat(:m,:p,:7) ,␣
↪xnobs=xnobs(:m,:p) , nobias=nobias                 )
```

Examples:

ex1_comp_unistat.F90

ex2_comp_unistat.F90

comp_unistat_miss()

Purpose:

comp_unistat_miss() computes estimates of univariate statistics from a data array, *X*. *X* can be a vector, a matrix or a tri- or four-dimensional array of data, possibly containing missing values.

The subroutine computes the univariate statistics with only one pass through the data.

If all the data are not available at once, **comp_unistat_miss()** can operate on chunks of data.

On output, the argument *XSTAT* will contain the following statistics:

- *XSTAT(...,1)* contains the mean value of the data vector.
- *XSTAT(...,2)* contains the variance of the data vector.
- *XSTAT(...,3)* contains the standard deviation of the data vector.
- *XSTAT(...,4)* contains the coefficient of skewness of the data vector.
- *XSTAT(...,5)* contains the coefficient of kurtosis of the data vector.
- *XSTAT(...,6)* contains the minimum of the data vector.
- *XSTAT(...,7)* contains the maximum of the data vector.

comp_unistat_miss() computes unbiased estimates of variance and standard deviation. Unbiased estimates of skewness and kurtosis are computed only if the *NOBIAS* logical argument is used with the value `true`.

Synopsis:

```
call comp_unistat_miss( x(:n)          , first , last , xmiss , xstat(:7)     ␣
↪  , xnobs=xnobs        , nobias=nobias                 )
call comp_unistat_miss( x(:m,:n)       , first , last , xmiss , xstat(:m,:7) ␣
↪  , xnobs=xnobs(:m)    , nobias=nobias , dimvar=dimvar )
call comp_unistat_miss( x(:m,:p,:n)    , first , last , xmiss , xstat(:m,:p,
↪:7) , xnobs=xnobs(:m,:p) , nobias=nobias                )
call comp_unistat_miss( x(:m,:p:q,,:n) , first , last , xmiss , xstat(:m,:p,
↪:7) , xnobs=xnobs(:m,:p) , nobias=nobias                )
```

comp_mvs()

Purpose:

comp_mvs() computes estimates of means, variances and standard-deviations from a data array, *X*. *X* can be a vector, a matrix or a tri- or four-dimensional array of data.

The subroutine computes the basic statistics with only one pass through the data.

If all the data are not available at once, **comp_mvs()** can operate on chunks of data.

comp_mvs() computes biased estimates of variance and standard deviation.

Synopsis:

```
call comp_mvs( x(:n)          , first , last , xmean          , xvar
↪  , xstd          ,          xnobs=xnobs                )
call comp_mvs( x(:m,:n)       , first , last , xmean(:m)      , xvar(:m)
↪  , xstd(:m)      ,          xnobs=xnobs , dimvar=dimvar )
call comp_mvs( x(:m,:p,:n)    , first , last , xmean(:m,:p)   , xvar(:m,:p)
↪  , xstd(:m,:p)   ,          xnobs=xnobs                 )
call comp_mvs( x(:m,:p,:q,:n) , first , last , xmean(:m,:p,:q) , xvar(:m,:p,
↪:q) , xstd(:m,:p,:q) ,         xnobs=xnobs                )
call comp_mvs( x(:n)          , first , last , xmean          , xvar
↪  , xstd          , xmiss , xnobs=xnobs                   )
call comp_mvs( x(:m,:n)       , first , last , xmean(:m)      , xvar(:m)
↪  , xstd(:m)      , xmiss , xnobs=xnobs(:m), dimvar=dimvar )
call comp_mvs( x(:m,:p,:n)    , first , last , xmean(:m,:p)   , xvar(:m,:p)
↪  , xstd(:m,:p)   , xmiss , xnobs=xnobs(:m,:p)            )
call comp_mvs( x(:m,:p,:q,:n) , first , last , xmean(:m,:p,:q) , xvar(:m,:p,
↪:q) , xstd(:m,:p,:q) , xmiss , xnobs=xnobs(:m,:p,:q)        )
```

Examples:

ex1_comp_mvs.F90

ex2_comp_mvs.F90

comp_mvs_miss()

Purpose:

comp_mvs_miss() computes estimates of means, variances and standard-deviations from a data array, *X*. *X* can be a vector, a matrix or a tri- or four-dimensional array of data, possibly containing missing values.

The subroutine computes the basic statistics with only one pass through the data.

If all the data are not available at once, **comp_mvs_miss()** can operate on chunks of data.

comp_mvs_miss() computes biased estimates of variance and standard deviation.

Synopsis:

```
call comp_mvs_miss( x(:n)          , first , last , xmean          , xvar
↪       , xstd          , xmiss , xnobs=xnobs                  )
call comp_mvs_miss( x(:m,:n)       , first , last , xmean(:m)      ,␣
↪xvar(:m)       , xstd(:m)       , xmiss , xnobs=xnobs(:m), dimvar=dimvar )
call comp_mvs_miss( x(:m,:p,:n)    , first , last , xmean(:m,:p)   , xvar(:m,
↪:p)    , xstd(:m,:p)    , xmiss , xnobs=xnobs(:m,:p)            )
call comp_mvs_miss( x(:m,:p,:q,:n) , first , last , xmean(:m,:p,:q) , xvar(:m,
↪:p,:q) , xstd(:m,:p,:q) , xmiss , xnobs=xnobs(:m,:p,:q)          )
```

update_mvs()

Purpose:

update_mvs() computes sample mean and corrected sum of squares for a sample of size `XNOBS+XNOBS2` given the means and corrected sums of squares for two subsamples of size *XNOBS* and *XNOBS2* as output by a call to *comp_mvs()* when *LAST* = `false` on the two subsamples separetely.

The sample means, standard-deviations for the sample of size `XNOBS+XNOBS2` may be obtained by a final call to *comp_mvs()* with *LAST* = `true` and no observations.

One possible application of **update_mvs()** is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares computed for each

subsample independently using *comp_mvs()*. The means and corrected sums of squares for the original sample can then be calculated using **update_mvs()**. Finally, the means, variances and standard-deviations for the original sample can be computed by a final call to *comp_mvs()* with *LAST* = `true` and no observations.

Synopsis:

```
call update_mvs( xmean           , xvar           , xnobs , xmean2           ,
↪ xvar2           , xnobs2 )
call update_mvs( xmean(:m)       , xvar(:m)       , xnobs , xmean2(:m)       ,
↪ xvar2(:m)       , xnobs2 )
call update_mvs( xmean(:m,:p)    , xvar(:m,:p)    , xnobs , xmean2(:m,:p)    ,
↪ xvar2(:m,:p)    , xnobs2 )
call update_mvs( xmean(:m,:p,:q) , xvar(:m,:p,:q) , xnobs , xmean2(:m,:p,:q) ,
↪ xvar2(:m,:p,:q) , xnobs2 )
```

comp_mvs_grp()

Purpose:

comp_mvs_grp() computes estimates of means, variances and standard-deviations by groups from a data array, *X*. *X* can be a vector, a matrix or a tri- or four-dimensional array of data.

The subroutine computes the basic statistics by groups with only one pass through the data.

If all the data are not available at once, **comp_mvs_grp()** can operate on chunks of data.

Synopsis:

```
call comp_mvs_grp( x(:n)          , first , last , ngrp , ind(:n) , xmean_
↪grp(:ngrp)          , xstd_grp(:ngrp)          , xn_grp(:ngrp)              ␣
↪   )
call comp_mvs_grp( x(:m,:n)       , first , last , ngrp , ind(:n) ,␣
↪xmean_grp(:m,:ngrp)       , xstd_grp(:m,:ngrp)       , xn_grp(:ngrp),␣
↪dimvar=dimvar  )
call comp_mvs_grp( x(:m,:p,:n)    , first , last , ngrp , ind(:n) , xmean_
↪grp(:m,:p,:ngrp)    , xstd_grp(:m,:p,:ngrp)    , xn_grp(:ngrp)              ␣
↪   )
call comp_mvs_grp( x(:m,:p,:q,:n) , first , last , ngrp , ind(:n) , xmean_
↪grp(:m,:p,:q,:ngrp) , xstd_grp(:m,:p,:q,:ngrp) , xn_grp(:ngrp)              ␣
↪   )
call comp_mvs_grp( x(:n)          , first , last , ngrp , ind(:n) , xmean_
↪grp(:ngrp)          , xstd_grp(:ngrp)          , xn_grp(:ngrp)          ,␣
↪xmiss                )
call comp_mvs_grp( x(:m,:n)       , first , last , ngrp , ind(:n) , xmean_
↪grp(:m,:ngrp)       , xstd_grp(:m,:ngrp)       , xn_grp(:m,:ngrp)       ,␣
↪xmiss, dimvar=dimvar )
call comp_mvs_grp( x(:m,:p,:n)    , first , last , ngrp , ind(:n) , xmean_
↪grp(:m,:p,:ngrp)    , xstd_grp(:m,:p,:ngrp)    , xn_grp(:m,:p,:ngrp)    ,␣
↪xmiss                )
call comp_mvs_grp( x(:m,:p,:q,:n) , first , last , ngrp , ind(:n) , xmean_
↪grp(:m,:p,:q,:ngrp) , xstd_grp(:m,:p,:q,:ngrp) , xn_grp(:m,:p,:q,:ngrp) ,␣
↪xmiss                )
```

comp_mvs_grp_miss()

Purpose:

comp_mvs_grp_miss() computes estimates of means, variances and standard-deviations by groups from a data array, *X*. *X* can be a vector, a matrix or a tri- or four-dimensional array of data, possibly containing missing values.

The subroutine computes the basic statistics by groups with only one pass through the data.

If all the data are not available at once, **comp_mvs_grp_miss()** can operate on chunks of data.

Synopsis:

```
call comp_mvs_grp_miss( x(:n)          , first , last , ngrp , ind(:n) ,␣
↪xmean_grp(:ngrp)          , xstd_grp(:ngrp)          , xn_grp(:ngrp)    ␣
↪     , xmiss                )
call comp_mvs_grp_miss( x(:m,:n)       , first , last , ngrp , ind(:n) ,␣
↪xmean_grp(:m,:ngrp)       , xstd_grp(:m,:ngrp)       , xn_grp(:m,:ngrp) ␣
↪     , xmiss, dimvar=dimvar )
call comp_mvs_grp_miss( x(:m,:p,:n)    , first , last , ngrp , ind(:n) ,␣
↪xmean_grp(:m,:p,:ngrp)    , xstd_grp(:m,:p,:ngrp)    , xn_grp(:m,:p,:ngrp) ␣
↪   , xmiss                )
call comp_mvs_grp_miss( x(:m,:p,:q,:n) , first , last , ngrp , ind(:n) ,␣
↪xmean_grp(:m,:p,:q,:ngrp) , xstd_grp(:m,:p,:q,:ngrp) , xn_grp(:m,:p,:q,
↪:ngrp) , xmiss                )
```

update_mvs_grp()

Purpose:

update_mvs_grp() computes sample mean and corrected sum of squares by groups for a sample of size `sum(XN_GRP)+sum(XN_GRP2)` given the means and corrected sums of squares by groups for two subsamples of size `sum(XN_GRP)` and `sum(XN_GR2P)`, as output by a call to *comp_mvs_grp()* when *LAST* = `false` on the two subsamples separetely.

The sample means, standard-deviations by groups for the sample of size `sum(XN_GRP)+sum(XN_GRP2)` may be obtained by a final call to *comp_mvs_grp()* with *LAST* = `true` and no observations.

One possible application of **update_mvs_grp()** is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares by groups computed for each subsample independently using *comp_mvs_grp()*. The means and corrected sums of squares by groups for the original sample can then be calculated using **update_mvs_grp()**. Finally, the means, variances and standard-deviations by groups for the original sample can be computed by a final call to *comp_mvs_grp()* with *LAST* = `true` and no observations.

Synopsis:

```
call update_mvs_grp( xmean_grp(:n)          , xstd_grp(:n)          , xn_
↪grp(:n)          , xmean_grp2(:n)          , xstd_grp2(:n)          , xn_
↪grp2(:n)          )
call update_mvs_grp( xmean_grp(:m,:n)       , xstd_grp(:m,:n)       , xn_
↪grp(:n)          , xmean_grp2(:m,:n)       , xstd_grp2(:m,:n)       , xn_
↪grp2(:n)          )
call update_mvs_grp( xmean_grp(:m,:p,:n)    , xstd_grp(:m,:p,:n)    , xn_
↪grp(:n)          , xmean_grp2(:m,:p,:n)    , xstd_grp2(:m,:p,:n)    , xn_
↪grp2(:n)          )
call update_mvs_grp( xmean_grp(:m,:p,:q,:n) , xstd_grp(:m,:p,:q,:n) , xn_
↪grp(:n)          , xmean_grp2(:m,:p,:q,:n) , xstd_grp2(:m,:p,:q,:n) , xn_
↪grp2(:n)          )
call update_mvs_grp( xmean_grp(:m,:n)       , xstd_grp(:m,:n)       , xn_
↪grp(:m,:n)       , xmean_grp2(:m,:n)       , xstd_grp2(:m,:n)       , xn_
↪grp2(:m,:n)       )
call update_mvs_grp( xmean_grp(:m,:p,:n)    , xstd_grp(:m,:p,:n)    , xn_
↪grp(:m,:p,:n)    , xmean_grp2(:m,:p,:n)    , xstd_grp2(:m,:p,:n)    , xn_
↪grp2(:m,:p,:n)    )
```

```
call update_mvs_grp( xmean_grp(:m,:p,:q,:n) , xstd_grp(:m,:p,:q,:n) , xn_
↪grp(:m,:p,:q,:n) , xmean_grp2(:m,:p,:q,:n) , xstd_grp2(:m,:p,:q,:n) , xn_
↪grp2(:m,:p,:q,:n) )
```

update_mvs_grp_miss()

Purpose:

update_mvs_grp_miss() computes sample mean and corrected sum of squares by groups for a sample of size `sum(XN_GRP)+sum(XN_GRP2)`, possibly containing missing values, given the means and corrected sums of squares by groups for two subsamples of size `sum(XN_GRP)` and `sum(XN_GR2P)`, as output by a call to *comp_mvs_grp_miss()* when *LAST* = `false` on the two subsamples separetely.

The sample means, standard-deviations by groups for the sample of size `sum(XN_GRP)+sum(XN_GRP2)` may be obtained by a final call to *comp_mvs_grp_miss()* with *LAST* = `true` and no observations.

One possible application of **update_mvs_grp_miss()** is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares by groups computed for each subsample independently using *comp_mvs_grp_miss()*. The means and corrected sums of squares by groups for the original sample can then be calculated using **update_mvs_grp_miss()**. Finally, the means, variances and standard-deviations by groups for the original sample can be computed by a final call to *comp_mvs_grp_miss()* with *LAST* = `true` and no observations.

Synopsis:

```
call update_mvs_grp_miss( xmean_grp(:n)           , xstd_grp(:n)           ,␣
↪xn_grp(:n)           , xmean_grp2(:n)           , xstd_grp2(:n)           , xn_
↪grp2(:n)           )
call update_mvs_grp_miss( xmean_grp(:m,:n)        , xstd_grp(:m,:n)        ,␣
↪xn_grp(:m,:n)        , xmean_grp2(:m,:n)        , xstd_grp2(:m,:n)        , xn_
↪grp2(:m,:n)        )
call update_mvs_grp_miss( xmean_grp(:m,:p,:n)     , xstd_grp(:m,:p,:n)     ,␣
↪xn_grp(:m,:p,:n)     , xmean_grp2(:m,:p,:n)     , xstd_grp2(:m,:p,:n)     , xn_
↪grp2(:m,:p,:n)     )
call update_mvs_grp_miss( xmean_grp(:m,:p,:q,:n) , xstd_grp(:m,:p,:q,:n) ,␣
↪xn_grp(:m,:p,:q,:n) , xmean_grp2(:m,:p,:q,:n) , xstd_grp2(:m,:p,:q,:n) , xn_
↪grp2(:m,:p,:q,:n) )
```

comp_anoma()

Purpose:

comp_anoma() computes anomalies (e.g., differences with the mean) or standardized anomalies from a data array *X*. *X* can be a vector, a matrix or a tridimensional array.

Synopsis:

```
call comp_anoma( x(:n)       ,          xmean        , xstd=xstd              ␣
↪          )
call comp_anoma( x(:m,:n)    ,          xmean(:m)    , xstd=xstd(:m)    ,␣
↪dimvar=dimvar )
call comp_anoma( x(:m,:p,:n) ,          xmean(:m,:p) , xstd=xstd(:m,:p)       ␣
↪          )
```

comp_anoma_miss()

Purpose:

comp_anoma_miss() computes anomalies (e.g., differences with the mean) or standardized anomalies from a data array *X*, possibly containing missing values. *X* can be a vector, a matrix or a tridimensional array.

Synopsis:

```
call comp_anoma( x(:n)       , xmiss , xmean        , xstd=xstd              ␣
↪          )
call comp_anoma( x(:m,:n)    , xmiss , xmean(:m)    , xstd=xstd(:m)    ,␣
↪dimvar=dimvar )
call comp_anoma( x(:m,:p,:n) , xmiss , xmean(:m,:p) , xstd=xstd(:m,:p)       ␣
↪          )
```

comp_anoma_grp()

Purpose:

comp_anoma_grp() computes anomalies (e.g., differences with the mean) or standardized anomalies by groups from a data array *X*. *X* can be a vector, a matrix or a tridimensional array.

Synopsis:

```
call comp_anoma_grp( x(:n)       , ngrp , ind(:n) , xmean_grp(:ngrp)       ,␣
↪xstd_grp=xstd_grp(:ngrp)                        )
call comp_anoma_grp( x(:m,:n)    , ngrp , ind(:n) , xmean_grp(:m,:ngrp)    ,␣
↪xstd_grp=xstd_grp(:m,:ngrp)    , dimvar=dimvar )
call comp_anoma_grp( x(:m,:p,:n) , ngrp , ind(:n) , xmean_grp(:m,:p,:ngrp) ,␣
↪xstd_grp=xstd_grp(:m,:p,:ngrp)                  )
```

comp_anoma_grp_miss()

Purpose:

comp_anoma_grp_miss() computes anomalies (e.g., differences with the mean) or standardized anomalies by groups from a data array *X*, possibly containing missing values. *X* can be a vector, a matrix or a tridimensional array.

Synopsis:

```
call comp_anoma_grp_miss( x(:n)       , ngrp , ind(:n) , xmiss , xmean_
↪grp(:ngrp)       , xstd_grp=xstd_grp(:ngrp)                        )
call comp_anoma_grp_miss( x(:m,:n)    , ngrp , ind(:n) , xmiss , xmean_grp(:m,
↪:ngrp)    , xstd_grp=xstd_grp(:m,:ngrp)    , dimvar=dimvar )
call comp_anoma_grp_miss( x(:m,:p,:n) , ngrp , ind(:n) , xmiss , xmean_grp(:m,
↪:p,:ngrp) , xstd_grp=xstd_grp(:m,:p,:ngrp)                  )
```

comp_composite()

Purpose:

Purpose:

comp_composite() computes a composite analysis from an array of data *X* [Terray_etal:2003]. The array argument *X* can be a vector, a matrix or a tridimensional array of data and **comp_composite_miss()** computes all the relevant statistics with one pass through the data.

Synopsis:

```
call comp_composite( x(:n)       , first , last , ngrp , ind(:n) , xmean      ␣
↪  , xstd        , xn        , xmean_grp(:ngrp)       , xstd_grp(:ngrp)     ␣
↪ , xn_grp(:ngrp) ,                                xcomp=xcomp(:ngrp)       ,␣
↪u=u(:ngrp)       , prob=prob(:ngrp)       , utest=utest )
call comp_composite( x(:m,:n)    , first , last , ngrp , ind(:n) , xmean(:m)  ␣
↪  , xstd(:m)    , xn        , xmean_grp(:m,:ngrp)    , xstd_grp(:m,:ngrp)  ␣
↪ , xn_grp(:ngrp) ,              dimvar=dimvar , xcomp=xcomp(:m,:ngrp)    ,␣
↪u=u(:m,:ngrp)    , prob=prob(:m,:ngrp)    , utest=utest )
call comp_composite( x(:m,:p,:n) , first , last , ngrp , ind(:n) , xmean(:m,
↪:p) , xstd(:m,:p) , xn        , xmean_grp(:m,:p,:ngrp) , xstd_grp(:m,:p,
```

```
↪:ngrp) , xn_grp(:ngrp) ,                                    xcomp=xcomp(:m,:p,
↪:ngrp) , u=u(:m,:p,:ngrp) , prob=prob(:m,:p,:ngrp) , utest=utest )
call comp_composite( x(:n)       , first , last , ngrp , ind(:n) , xmean
↪  , xstd        , xn        , xmean_grp(:ngrp)       , xstd_grp(:ngrp)
↪ , xn_grp(:ngrp)       , xmiss ,                 xcomp=xcomp(:ngrp)       ,
↪u=u(:ngrp)       , prob=prob(:ngrp)       , utest=utest )
call comp_composite( x(:m,:n)    , first , last , ngrp , ind(:n) , xmean(:m)
↪  , xstd(:m)    , xn(:m)    , xmean_grp(:m,:ngrp)    , xstd_grp(:m,:ngrp)
↪ , xn_grp(:m,:ngrp)    , xmiss , dimvar=dimvar , xcomp=xcomp(:m,:ngrp)    ,
↪u=u(:m,:ngrp)    , prob=prob(:m,:ngrp)    , utest=utest )
call comp_composite( x(:m,:p,:n) , first , last , ngrp , ind(:n) , xmean(:m,
↪:p) , xstd(:m,:p) , xn(:m,:p) , xmean_grp(:m,:p,:ngrp) , xstd_grp(:m,:p,
↪:ngrp) , xn_grp(:m,:p,:ngrp) , xmiss ,                  xcomp=xcomp(:m,:p,
↪:ngrp) , u=u(:m,:p,:ngrp) , prob=prob(:m,:p,:ngrp) , utest=utest )
```

comp_composite_miss()

Purpose:

comp_composite_miss() computes a composite analysis from an array of data *X* [Terray_etal:2003]. The array argument *X* can be a vector, a matrix or a tridimensional array of data, possibly containing missing data, and **comp_composite_miss()** computes all the relevant statistics with one pass through the data.

Synopsis:

```
call comp_composite_miss( x(:n)       , first , last , ngrp , ind(:n) , xmean
↪       , xstd        , xn        , xmean_grp(:ngrp)       , xstd_grp(:ngrp)
↪      , xn_grp(:ngrp)       , xmiss ,                 xcomp=xcomp(:ngrp)
↪   , u=u(:ngrp)       , prob=prob(:ngrp)       , utest=utest )
call comp_composite_miss( x(:m,:n)    , first , last , ngrp , ind(:n)
↪, xmean(:m)    , xstd(:m)    , xn(:m)    , xmean_grp(:m,:ngrp)    ,
↪xstd_grp(:m,:ngrp)    , xn_grp(:m,:ngrp)    , xmiss , dimvar=dimvar ,
↪xcomp=xcomp(:m,:ngrp)    , u=u(:m,:ngrp)    , prob=prob(:m,:ngrp)    ,
↪utest=utest )
call comp_composite_miss( x(:m,:p,:n) , first , last , ngrp , ind(:n)
↪, xmean(:m,:p) , xstd(:m,:p) , xn(:m,:p) , xmean_grp(:m,:p,:ngrp) ,
↪xstd_grp(:m,:p,:ngrp) , xn_grp(:m,:p,:ngrp) , xmiss ,
↪xcomp=xcomp(:m,:p,:ngrp) , u=u(:m,:p,:ngrp) , prob=prob(:m,:p,:ngrp) ,
↪utest=utest )
```

valmed()

Purpose:

valmed() finds the medians of a `n`-element vector or of the column vectors of a matrix.

valmed() uses a modified quicksort algorithm.

Synopsis:

```
median     = valmed( x(:n)    )
median(:m) = valmed( x(:n,:m) )
```

5.25 MODULE Mul_Stat_Procedures

Module *Mul_Stat_Procedures* exports subroutines and functions for multivariate statistical computations. More specifically, routines for performing correlation/regression analysis, Principal Component Analysis (e.g., Empirical Orthogonal analysis in climate research), Maximum Correlation Analysis (MCA) on datasets with or without missing

values are included in this module. For an introduction on these different methods, see for example [Jolliffe:2002] [Jackson:2003] [vonStorch_Zwiers:2002] [Bretherton_etal:1992]. In addition, a large variety of orthogonal rotation methods of a partial PCA model are also provided to allow a better and accurate exploration of the main spatial patterns and/or low- or high frequency modes structuring huge multivariate geophysical datasets [Jolliffe:2002] [Jackson:2003] [Jennrich:1970] [Clarkson_Jennrich:1988] [Arbuckle_Friendly:1977] [Wills_etal:2018].

Please note that routines provided in this module apply only to real data of kind **stnd**. The real kind type **stnd** is defined in module *Select_Parameters*.

In order to use one of these routines, you must include an appropriate `use Mul_Stat_Procedures` or `use Statpack` statement in your Fortran program, like:

```
use Mul_Stat_Procedures, only: comp_cor
```

or:

```
use Statpack, only: comp_cor
```

Here is the list of the public routines exported by module *Mul_Stat_Procedures*:

comp_cor()

Purpose:

comp_cor() computes estimates of means, variances, correlation and regression coefficients from two data arrays *X* and *Y*.

comp_cor() computes the basic univariate statistics and correlation coefficients with only one pass through the data and is efficient on huge datasets.

Moreover, **comp_cor()** also allows out-of-core processing of the data at the user option.

For more details on correlation and regression analysis, see Chapter 8 of [vonStorch_Zwiers:2002].

Synopsis:

```
call comp_cor( x(:n)       , y(:n)    , first , last , xstat(:2)       ,␣
↪ystat(:2)    , xycor        , xyn ,                                    z=z   ␣
↪     , prob=prob       , ndf_max=ndf_max , cortest=cortest , cov=cov )
call comp_cor( x(:m,:n)    , y(:n)    , first , last , xstat(:m,:2)    ,
↪ ystat(:2)    , xycor(:m)    , xyn , dimvar=dimvar ,                    ␣
↪z=z(:m)    , prob=prob(:m)    , ndf_max=ndf_max , cortest=cortest , cov=cov␣
↪)
call comp_cor( x(:m,:p,:n) , y(:n)    , first , last , xstat(:m,:p,:2) ,
↪ ystat(:2)    , xycor(:m,:p) , xyn ,                                    ␣
↪z=z(:m,:p) , prob=prob(:m,:p) , ndf_max=ndf_max , cortest=cortest , cov=cov␣
↪)
call comp_cor( x(:m,:n)    , y(:p,:n) , first , last , xstat(:m,:2)    ,
↪ ystat(:p,:2) , xycor(:m,:p) , xyn , dimvar=dimvar , dimvary=dimvary ,␣
↪z=z(:m,:p) , prob=prob(:m,:p) , ndf_max=ndf_max , cortest=cortest , cov=cov␣
↪)
```

Examples:

ex1_comp_cor.F90

ex2_comp_cor.F90

comp_cor_miss()

Purpose:

comp_cor_miss() computes estimates of means, variances, correlation and regression coefficients from two data arrays *X* and *Y*, possibly containing missing values.

comp_cor_miss() computes the basic univariate statistics and correlation coefficients with only one pass through the data and is efficient on huge datasets.

The means and standard-deviations of *X* and *Y* are computed from all valid data. The correlation coefficients are based on these univariate statistics and on all valid pairs of observations.

Moreover, **comp_cor_miss()** also allows out-of-core processing of the data at the user option.

For more details on correlation and regression analysis, see Chapter 8 of [vonStorch_Zwiers:2002].

Synopsis:

```
call comp_cor_miss( x(:n)       , y(:n)    , first , last , xstat(:4)       ,␣
↪ystat(:4)    , xycor(:4)       , xymiss ,                                    ␣
↪z=z        , prob=prob         , ndf_max=ndf_max , cov=cov )
call comp_cor_miss( x(:m,:n)    , y(:n)    , first , last , xstat(:m,:4)    ,␣
↪ystat(:4)    , xycor(:m,:4)    , xymiss , dimvar=dimvar ,                    ␣
↪z=z(:m)    , prob=prob(:m)     , ndf_max=ndf_max , cov=cov )
call comp_cor_miss( x(:m,:p,:n) , y(:n)    , first , last , xstat(:m,:p,:4) ,␣
↪ystat(:4)    , xycor(:m,:p,:4) , xymiss ,                                    ␣
↪z=z(:m,:p) , prob=prob(:m,:p) , ndf_max=ndf_max , cov=cov )
call comp_cor_miss( x(:m,:n)    , y(:p,:n) , first , last , xstat(:m,:4)    ,␣
↪ystat(:p,:4) , xycor(:m,:p,:4) , xymiss , dimvar=dimvar , dimvary=dimvary ,␣
↪z=z(:m,:p) , prob=prob(:m,:p) , ndf_max=ndf_max , cov=cov )
```

Examples:

ex1_comp_cor_miss.F90

ex2_comp_cor_miss.F90

comp_cor_miss2()

Purpose:

comp_cor_miss2() computes estimates of means, variances, correlation and regression coefficients from two data arrays *X* and *Y*, possibly containing missing values.

comp_cor_miss2() computes the basic univariate statistics and correlation coefficients with only one pass through the data and is efficient on huge datasets.

The univariate and bivariate statistics are computed from all valid pairs of observations. This differs from the method used in *comp_cor_miss()*.

Moreover, **comp_cor_miss2()** also allows out-of-core processing of the data at the user option.

For more details on correlation and regression analysis, see Chapter 8 of [vonStorch_Zwiers:2002].

Synopsis:

```
call comp_cor_miss2( x(:n)       , y(:n) , first , last , xstat(:2)       ,␣
↪ystat(:2)       , xycor         , xyn         , xymiss ,                  z=z ␣
↪      , prob=prob         , ndf_max=ndf_max )
call comp_cor_miss2( x(:m,:n)    , y(:n) , first , last , xstat(:m,:2)    ,
↪ ystat(:m,:2)    , xycor(:m)    , xyn(:m)     , xymiss , dimvar=dimvar ,␣
↪z=z(:m)    , prob=prob(:m)     , ndf_max=ndf_max )
call comp_cor_miss2( x(:m,:p,:n) , y(:n) , first , last , xstat(:m,:p,:2) ,
↪ ystat(:m,:p,:2) , xycor(:m,:p) , xyn(:m,:p) , xymiss ,                    ␣
↪z=z(:m,:p) , prob=prob(:m,:p) , ndf_max=ndf_max )
```

Examples:

ex1_comp_cor_miss2.F90

permute_cor()

Purpose:

permute_cor() performs permutation tests of a correlation coefficients between two data arrays *Y* and *X*.

permute_cor() is parallelized if OpenMP is used.

For more details and algorithms see Chapter 8 of [vonStorch_Zwiers:2002] and also [Noreen:1989].

Synopsis:

```
call permute_cor( x(:n)    , y(:n) , xstat(:2)    , ystat(:2) , xycor     ,␣
↪prob ,                      nrep=nrep , initseed=initseed )
call permute_cor( x(:m,:n) , y(:n) , xstat(:m,:2) , ystat(:2) , xycor(:m) ,␣
↪prob(:m) , dimvar=dimvar , nrep=nrep , initseed=initseed )
```

Examples:

ex1_permute_cor.F90

ex2_permute_cor.F90

phase_scramble_cor()

Purpose:

phase_scramble_cor() performs phase-scrambled bootstrap tests of correlation coefficients between two data arrays *Y* and *X*.

phase_scramble_cor() is parallelized if OpenMP is used.

For more details and algorithms see [Theiler_etal:1992] [Ebisuzaki:1997] [Braun_Kulperger:1997] [Davison_Hinkley:1997].

Synopsis:

```
call phase_scramble_cor( x(:n)    , y(:n) , xstat(:2)    , ystat(:2) , xycor␣
↪    , prob     ,                  nrep=nrep , method=method , norm=norm ,␣
↪initseed=initseed )
call phase_scramble_cor( x(:m,:n) , y(:n) , xstat(:m,:2) , ystat(:2) ,
↪ xycor(:m) , prob(:m) , dimvar=dimvar , nrep=nrep , method=method ,␣
↪norm=norm , initseed=initseed )
```

Examples:

ex1_phase_scramble_cor.F90

ex2_phase_scramble_cor.F90

bootstrap_cor()

Purpose:

bootstrap_cor() performs moving block bootstrap tests of correlation coefficients between two data arrays *Y* and *X*.

bootstrap_cor() is parallelized if OpenMP is used.

For more details and algorithms see [Davison_Hinkley:1997].

Synopsis:

```
call bootstrap_cor( x(:n)     , y(:n) , xstat(:2)     , xycor(:2) , prob     ,
↪                  nrep=nrep , initseed=initseed , periodicity=periodicity ,␣
↪season=season , block_size=block_size )
call bootstrap_cor( x(:m,:n) , y(:n) , xstat(:m,:2) , xycor(:m) , prob(:m) ,
↪ dimvar=dimvar , nrep=nrep , initseed=initseed , periodicity=periodicity ,␣
↪season=season , block_size=block_size )
```

update_cor()

Purpose:

update_cor() computes sample means and corrected sums of squares and cross-products for a sample of size *XYN*+*XYN2* given the means and corrected sums of squares and cross-products for two subsamples of size *XYN* and *XYN2* as output by a call to *comp_cor()* when *LAST* = `false` on the two subsamples separately.

The sample means, variances and coefficient correlations for the sample of size *XYN*+*XYN2* may be obtained by a call to *comp_cor()* with *LAST* = `true` and no observations.

One possible application of this subroutine is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares and cross-products computed for each subsample independently using *comp_cor()*. The means and corrected sums of squares and cross-products for the original sample can then be calculated using **update_cor()**. The means, variances and correlation coefficients for the original sample can be computed by a final call to *comp_cor()* with *LAST* = `true`.

Synopsis:

```
call update_cor( xstat(:2)       , ystat(:2) , xycor        , xyn ,␣
↪xstat2(:2)       , ystat2(:2) , xycor2        , xyn2 )
call update_cor( xstat(:m,:2)    , ystat(:2) , xycor(:m)    , xyn , xstat2(:m,
↪:2)    , ystat2(:2) , xycor2(:m)    , xyn2 )
call update_cor( xstat(:m,:p,:2) , ystat(:2) , xycor(:m,:p) , xyn , xstat2(:m,
↪:p,:2) , ystat2(:2) , xycor2(:m,:p) , xyn2 )
```

update_cor_miss2()

Purpose:

update_cor_miss2() computes sample means and corrected sums of squares and cross-products for a sample of size *XYN*+*XYN2*, possibly containing missing values, given the means and corrected sums of squares and cross-products for two subsamples of size *XYN* and *XYN2* as output by a call to *comp_cor_miss2()* when *LAST* = `false` on the two subsamples separately.

The sample means, variances and coefficient correlations for the sample of size *XYN*+*XYN2* may be obtained by a call to *comp_cor()* with *LAST* = `true` and no observations.

One possible application of this subroutine is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares and cross-products computed for each subsample independently using *comp_cor_miss2()*. The means and corrected sums of squares and cross-products for the original sample can then be calculated using **update_cor_miss2()**. The means, variances and correlation coefficients for the original sample can be computed by a final call to *comp_cor_miss2()* with *LAST* = `true`.

Synopsis:

```
call update_cor_miss2      ( xstat(:2)       , ystat(:2)       , xycor       ␣
↪, xyn        , xstat2(:2)       , ystat2(:2)       , xycor2        , xyn2   ␣
↪     )
call update_cor_miss2( xstat(:m,:2)    , ystat(:m,:2)    , xycor(:m)    ,␣
↪xyn(:m)    , xstat2(:m,:2)    , ystat2(:m,:2)    , xycor2(:m)    , xyn2(:m)␣
↪   )
```

```
call update_cor_miss2( xstat(:m,:p,:2) , ystat(:m,:p,:2) , xycor(:m,:p) ,␣
↪xyn(:m,:p) , xstat2(:m,:p,:2) , ystat2(:m,:p,:2) , xycor2(:m,:p) , xyn2(:m,
↪:p) )
```

comp_cormat()

Purpose:

comp_cormat() computes estimates of means and variance-covariance or correlation matrix (eventually in packed form in the output array argument *XCORP*) from a data matrix *X*.

comp_cormat() computes the means and correlation matrix with only one pass through the data and is efficient on huge datasets.

Moreover, **comp_cormat()** also allows out-of-core processing of the data at the user option.

Synopsis:

```
call comp_cormat( x(:m,:n) , first , last , xmean(:m) , xcor(:m,:m)          ,␣
↪xn , dimvar=dimvar , xstd=xstd(:m) , cov=cov , fill=fill , failure=failure )
call comp_cormat( x(:m,:n) , first , last , xmean(:m) , xcorp(:(m*(m+1))/2) ,␣
↪xn , dimvar=dimvar , xstd=xstd(:m) , cov=cov ,              failure=failure )
```

Examples:

ex1_comp_cormat.F90

ex2_comp_cormat.F90

comp_cormat_miss()

Purpose:

comp_cormat_miss() computes estimates of means and variance-covariance or correlation matrix (eventually in packed form in the output array argument *XCORP*) from a data matrix *X*, possibly containing missing values.

The means and standard-deviations of the data matrix *X* are computed from all valid data. The correlation coefficients are based on these univariate statistics and on all valid pairs of observations.

comp_cormat_miss() computes the means and correlation matrix with only one pass through the data and is efficient on huge datasets.

Moreover, **comp_cormat_miss()** also allows out-of-core processing of the data at the user option.

Synopsis:

```
call comp_cormat_miss( x(:m,:n) , first , last , xmean(:m,:2) , xcor(:m,:m)␣
↪        , xn(:(m*(m+1))/2,:3) , xmiss , dimvar=dimvar , xstd=xstd(:m) ,␣
↪cov=cov , fill=fill , failure=failure )
call comp_cormat_miss( x(:m,:n) , first , last , xmean(:m,:2) ,␣
↪xcorp(:(m*(m+1))/2) , xn(:(m*(m+1))/2,:3) , xmiss , dimvar=dimvar ,␣
↪xstd=xstd(:m) , cov=cov ,              failure=failure )
```

Examples:

ex1_comp_cormat_miss.F90

ex2_comp_cormat_miss.F90

comp_eof()

Purpose:

comp_eof() computes estimates of Empirical Orthogonal Functions (e.g. EOF, also known as Principal Component Analysis) from a data matrix *X* [vonStorch_Zwiers:2002] [Jolliffe:2002] [Jackson:2003] .

comp_eof() computes the Empirical Orthogonal Functions with only one pass through the data and allows out-of-core processing at the user option.

The eigenvectors of the covariance or correlation matrix are computed with the *eig_cmp2()* subroutine in module *EIG_Procedures*.

Finally, **comp_eof()** may be used in a call with no observations (e.g. with `size(X,3-DIMVAR) = 0`) in order to finish the computations with *LAST* = `true` when the total number of observations is unknown at the beginning of the computations.

Synopsis:

```
call comp_eof( x(:m,:n) , first , last , xeigval(:m) , xeigvec(:m,
↪:m) , xn , failure , dimvar=dimvar , cov=cov , sort=sort ,␣
↪maxiter=maxiter , xmean=xmean(:m) , xstd=xstd(:m) , xeigvar=xeigvar(:m) ,
↪ xcorp=xcorp(:(m*(m+1))/2) )
```

Examples:

ex1_comp_eof.F90

ex1_comp_ortho_rot_eof.F90

ex1_comp_filt_rot_pc.F90

ex1_comp_lfc_rot_pc.F90

ex1_comp_smooth_rot_pc.F90

comp_eof2()

Purpose:

comp_eof2() computes estimates of Empirical Orthogonal Functions (e.g. EOF, also known as Principal Component Analysis) from a data matrix *X* [vonStorch_Zwiers:2002].

comp_eof2() computes the Empirical Orthogonal Functions with only one pass through the data and allows out-of-core processing at the user option.

comp_eof2() computes all the eigenvalues, and, optionally, selected eigenvectors (by inverse iteration), of the covariance (or correlation matrix) from the data matrix.

Thus, **comp_eof2()** must be used instead of *comp_eof()* if you are only interested in the first few Empirical Orthogonal Functions, which explains the larger part of the variance of the data matrix *X* and for the processing of huge datasets.

The eigenvalues and (selected) eigenvectors of the covariance or correlation matrix are computed with the *eigval_cmp()* and *trid_inviter()* subroutines in module *EIG_Procedures*.

Finally, **comp_eof2()** may be used in a call with no observations (e.g. with `size(X,3-DIMVAR) = 0`) in order to finish the computations with *LAST* = `true` when the total number of observations is unknown at the beginning of the computations.

Synopsis:

```
call comp_eof2( x(:m,:n) , first , last , xeigval(:m) , xcorp(:(m*(m+1))/2) ,␣
↪xn , failure , dimvar=dimvar , cov=cov , savecor=savecor , maxiter=maxiter␣
↪, ortho=ortho , xmean=xmean(:m) , xstd=xstd(:m) , xeigvar=xeigvar(:m) ,␣
↪xeigvec=xeigvec(:m,:p) )
```

Examples:

ex1_comp_eof2.F90

comp_eof3()

Purpose:

comp_eof3() computes estimates of Empirical Orthogonal Functions (e.g. EOF, also known as Principal Component Analysis) from a data matrix `X` with `n` observations [vonStorch_Zwiers:2002].

comp_eof3() computes the matrix product

$\frac{1}{n}(X^T * X)$ or $\frac{1}{n}(X * X^T)$

from the data matrix `X`, and all the eigenvalues, and selected eigenvectors (by inverse iteration), of this matrix product.

The eigenvalues and (selected) eigenvectors of the matrix product are computed with the *eigval_cmp()* and *trid_inviter()* subroutines in module *EIG_Procedures*.

Synopsis:

```
call comp_eof3( x(:m,:n) , dimvar=dimvar , failure , xcorp=xcorp(:(m*(m+1))/
↪2) , xeigval=xeigval(:m,:2) , xeigvec=xeigvec(:m,:p) , maxiter=maxiter ,␣
↪ortho=ortho )
```

comp_eof_miss()

Purpose:

comp_eof_miss() computes estimates of Empirical Orthogonal Functions (e.g. EOF, also known as Principal Component Analysis) from a data matrix *X*, possibly containing missing values [vonStorch_Zwiers:2002].

The means and standard-deviations of the data matrix *X* are computed from all valid data. The covariance or correlation coefficients are based on these univariate statistics and on all valid pairs of observations. Finally, The eigenvectors and eigenvalues are estimated from these bivariate statistics.

comp_eof_miss() computes estimates of the Empirical Orthogonal Functions with only one pass through the data and allows out-of-core processing at the user option.

The eigenvectors of the covariance or correlation matrix are computed with the *eig_cmp2()* subroutine in module *EIG_Procedures*.

Finally, **comp_eof_miss()** may be used in a call with no observations (e.g. with `size(X,3-DIMVAR) = 0`) in order to finish the computations with *LAST* = `true` when the total number of observations is unknown at the beginning of the computations.

Synopsis:

```
call comp_eof_miss( x(:m,:n) , first , last , xeigval(:m,:2) , xeigvec(:m,
↪:m) , xcorp(:(m*(m+1))/2,:3) , xmiss , failure , dimvar=dimvar , cov=cov ,␣
↪sort=sort , maxiter=maxiter , xmean=xmean(:m) , xstd=xtsd(:m) )
```

comp_eof_miss2()

Purpose:

comp_eof_miss2() computes estimates of Empirical Orthogonal Functions (e.g. EOF, also known as Principal Component Analysis) from a data matrix *X*, possibly containing missing values [vonStorch_Zwiers:2002].

The means and standard-deviations of the data matrix *X* are computed from all valid data. The covariance or correlation coefficients are based on these univariate statistics and on all valid pairs of observations. Finally, The eigenvectors and eigenvalues are estimated from these bivariate statistics.

comp_eof_miss2() computes the Empirical Orthogonal Functions with only one pass through the data and allows out-of-core processing at the user option.

comp_eof_miss2() computes all the eigenvalues, and, optionally, selected eigenvectors (by inverse iteration), of the covariance (or correlation matrix) from the data matrix.

Thus, **comp_eof_miss2()** must be used instead of *comp_eof_miss()* if you are only interested in the first few Empirical Orthogonal Functions, which explains the larger part of the variance of the data matrix *X* and for the processing of huge datasets.

The eigenvalues and (selected) eigenvectors of the covariance or correlation matrix are computed with the *eigval_cmp()* and *trid_inviter()* subroutines in module *EIG_Procedures*.

Finally, **comp_eof_miss2()** may be used in a call with no observations (e.g. with `size(X,3-DIMVAR) = 0`) in order to finish the computations with *LAST* = `true` when the total number of observations is unknown at the beginning of the computations.

Synopsis:

```
call comp_eof_miss2( x(:m,:n) , first , last , xeigval(:m,:2) ,␣
↪xcorp(:(m*(m+1))/2,:4) , xmiss , failure , dimvar=dimvar , cov=cov ,
↪ maxiter=maxiter , ortho=ortho , xmean=xmean(:m) , xstd=xtsd(:m) ,␣
↪xeigvec=xeigvec(:m,:p) )
```

comp_eof_miss3()

Purpose:

comp_eof_miss3() computes estimates of Empirical Orthogonal Functions (e.g. EOF, also known as Principal Component Analysis) from a data matrix `X` with `n` observations, but possibly containing missing values [vonStorch_Zwiers:2002].

comp_eof_miss3() computes an estimate of the matrix product

$\frac{1}{n}(X^T * X)$ or $\frac{1}{n}(X * X^T)$

from the data matrix `X`, the associated matrix of incidence values, and all the eigenvalues, and selected eigenvectors (by inverse iteration), of this matrix product.

The estimate of the above matrix product is computed from all valid pairs of observations. The eigenvectors and eigenvalues are computed from these bivariate statistics.

The eigenvalues and, optionally, (selected) eigenvectors of the matrix product are computed with the *eigval_cmp()* and *trid_inviter()* subroutines in module *EIG_Procedures*.

Synopsis:

```
call comp_eof_miss3( x(:m,:n) , xmiss , dimvar , failure ,␣
↪xcorp=xcorp(:(m*(m+1))/2) , xincp=xincp(:(m*(m+1))/2) , xeigval=xeigval(:m,
↪:2) , xeigvec==xeigvec(:m,:p) , maxiter=maxiter , ortho=ortho )
```

comp_pc_eof()

Purpose:

comp_pc_eof() computes estimates of Principal Components (PC) from a data matrix and a set of eigenvectors derived from an EOF or PCA analysis.

If unnormalized PCs are desired, you must use argument *XSINGVAL* with all values set to one; however, in this case, do not use the optional argument *XPCCOR*, which will contain incorrect statistics if argument *XSINGVAL* is set to one.

Synopsis:

```
call comp_pc_eof( x(:m,:n) , xeigvec(:m,:p) , xsingval(:p) , xpc(:n,:p) ,␣
↪dimvar=dimvar , xmean=xmean(:m) , xstd=xtsd(:m) , xpccor=xpccor(:m,:p) )
```

Examples:

ex1_comp_eof.F90

ex1_comp_eof2.F90

ex1_comp_ortho_rot_eof.F90

ex1_comp_filt_rot_pc.F90

ex1_comp_lfc_rot_pc.F90

ex1_comp_smooth_rot_pc.F90

comp_ortho_rot_eof()

Purpose:

comp_ortho_rot_eof() performs an orthogonal rotation of a (partial) EOF model (e.g., a factor loading matrix) using a generalized orthomax criterion, including quartimax, varimax and equamax rotation methods.

For more details, see [Jennrich:1970] [Clarkson_Jennrich:1988] or [Jackson:2003].

Synopsis:

```
call comp_ortho_rot_eof( fac(:nv,:nf) , rot_fac(:nv,:nf) , orot(:nf,:nf)␣
↪, std_rot_fac(:nf) , failure , knorm=knorm , maxiter=maxiter , w=w ,␣
↪delta=delta )
```

Examples:

ex1_comp_ortho_rot_eof.F90

comp_smooth_rot_pc()

Purpose:

comp_smooth_rot_pc() performs an orthogonal rotation of a (partial) EOF model (e.g., the standardized Principal Component time series) by minimizing a smoothness criterion.

For more details, see [Arbuckle_Friendly:1977] [Jolliffe:2002].

Synopsis:

```
call comp_smooth_rot_pc( pc(:nobs,:nf) , std_pc(:nf) , rot_pc(:nobs,:nf)␣
↪, orot(:nf,:nf) , std_rot_pc(:nf) , failure , maxiter=maxiter , d=d ,␣
↪smooth=smooth )
```

Examples:

ex1_comp_smooth_rot_pc.F90

comp_lfc_rot_pc()

Purpose:

comp_lfc_rot_pc() performs an orthogonal rotation of a (partial) EOF model (e.g., the standardized Principal Component time series) towards low-frequency or high-frequency components using the eigenvectors of the covariance matrix between the standardized Principal Component time series filtered with a LOESS smoother.

For more details on this new orthogonal rotation method, see [Wills_etal:2018]. For information on the LOESS smoother used here for estimating the orthogonal rotation matrix applied to the standardized Principal Component time series, see [Cleveland:1979] [Cleveland_Devlin:1988] and the manual of the *Time_Series_Procedures* module.

Synopsis:

```
call comp_lfc_rot_pc( pc(:nobs,:nf) , std_pc(:nf) , nt , rot_pc(:nobs,:nf) ,␣
↪orot(:nf,:nf) , std_rot_pc(:nf) , failure , maxiter=maxiter , itdeg=itdeg ,␣
↪ntjump=ntjump , residual=residual , smooth=smooth(:nf) )
```

Examples:

ex1_comp_lfc_rot_pc.F90

comp_filt_rot_pc()

Purpose:

comp_filt_rot_pc() performs an orthogonal rotation of a (partial) EOF model (e.g., the standardized Principal Component time series) towards low-frequency, high-frequency or band-pass components using the eigenvectors of the covariance matrix between the standardized Principal Component time series filtered with a windowed FFT filter.

For more details on this new orthogonal rotation method, see [Wills_etal:2018]. For information on the windowed FFT filter used here for estimating the orthogonal rotation matrix applied to the standardized Principal Component time series, see [Iacobucci_Noullez:2005] and the manual of the *Time_Series_Procedures* module.

Synopsis:

```
call comp_filt_rot_pc( pc(:nobs,:nf) , std_pc(:nf) , pl , ph , rot_pc(:nobs,
↪:nf) , orot(:nf,:nf) , std_rot_pc(:nf) , failure , maxiter=maxiter ,␣
↪trend=trend , win=win , smooth=smooth(:nf) )
```

Examples:

ex1_comp_filt_rot_pc.F90

comp_mca()

Purpose:

comp_mca() performs a Maximum Covariance Analysis (MCA) or canonical covariance analysis between two data matrices *X* and *Y* [Bretherton_etal:1992] [Bjornsson_Venegas:1997].

comp_mca() computes the Singular Value Decomposition (SVD) of the m-by-n correlation (or covariance) matrix XYCOR between two data matrices X and Y. This SVD is written

$$XYCOR = U * S * V^T$$

where S is a min(m,n)-by-min(m,n) diagonal matrix, U is a m-by-min(m,n) orthogonal matrix, and V is a n-by-min(m,n) orthogonal matrix. The diagonal elements of S are the singular values of XYCOR, they are real and non-negative. The columns of U and V are, respectively, the left and right singular vectors of XYCOR.

The subroutine computes the basic univariate statistics and the correlation (or covariance) matrix with only one pass through the data and allows out-of-core processing for the computation of the correlation (or covariance) matrix at the user option.

The routine returns the singular values, the left and, optionally, the right singular vectors of the correlation (or covariance) matrix XYCOR between the two data matrices X and Y.

The singular values and singular vectors of the covariance or correlation matrix are computed with the *bd_cmp()*, *ortho_gen_bd()* (or *ortho_gen_q_bd()*) and *bd_svd()* subroutines in module *SVD_Procedures*.

Finally, **comp_mca()** may be used in a call with no observations (e.g. with size(X,3-DIMVARX) = 0) in order to finish the computations with *LAST* = true, when the total number of observations is unknown at the beginning of the computations.

Synopsis:

```
call comp_mca( x(:m,:n) , y(:p,:n) , first , last , xstat(:m,:2) ,␣
↪ystat(:p,:2) , xysingval(:min(m,p)) , xsingvec(:m,:p) , failure ,␣
↪dimvarx=dimvarx , dimvary=dimvary , cov=cov , sort=sort , maxiter=maxiter␣
↪, ysingvec=ysingvec(:p,:min(m,p)) , xysingvar=xysingvar(:min(m,p)) ,␣
↪xycor=xycor(:m,:p) )
```

Examples:

ex1_comp_mca.F90

comp_mca2()

Purpose:

comp_mca2() performs a Maximum Covariance Analysis (MCA) or canonical covariance analysis between two data matrices *X* and *Y* [Bretherton_etal:1992] [Bjornsson_Venegas:1997].

comp_mca2() computes a partial Singular Value Decomposition (SVD) of the m-by-n correlation (or covariance) matrix XYCOR between two data matrices X and Y. This partial SVD is written

$$XYCOR \simeq U(:m,:k) * S(:k,:k) * V(:n,:k)^T$$

where S is a k-by-k diagonal matrix, U is a m-by-k orthogonal matrix, and V is a n-by-k orthogonal matrix. The diagonal elements of S(:k,:k) are the largest singular values of XYCOR, they are real and non-negative. The columns of U(:m,:k) and V(:n,:k) are, respectively, the associated left and right singular vectors of XYCOR.

The subroutine computes the basic univariate statistics and the correlation (or covariance) matrix with only one pass through the data and allows out-of-core processing for the computation of the correlation (or covariance) matrix at the user option.

The routine returns all the singular values and, optionally, selected left and right singular vectors of the correlation (or covariance) matrix XYCOR between the two data matrices X and Y.

Thus, **comp_mca2()** must be used instead of *comp_mca()* if you are only interested in the first few singular triplets of XYCOR, which explains the larger part of the covariance or correlation between the data matrices *X* and *Y*, and for the processing of huge datasets.

The singular values and, optionally, selected singular vectors of the covariance or correlation matrix are computed (by inverse iteration) with the *svd_cmp()* and *bd_inviter2()* subroutines in module *SVD_Procedures*.

Finally, **comp_mca2()** may be used in a call with no observations (e.g. with size(X,3-DIMVARX) = 0) in order to finish the computations with *LAST* = true, when the total number of observations is unknown at the beginning of the computations.

Synopsis:

```
call comp_mca2( x(:m,:n) , y(:p,:n) , first , last , xstat(:m,:2) , ystat(:p,
↪:2) , xysingval(:min(m,p)) , xycor(:m,:p) , failure , dimvarx=dimvarx␣
↪, dimvary=dimvary , cov=cov , savecor=savecor  , maxiter=maxiter ,␣
↪ortho=ortho , xysingvar=xysingvar(:min(m,p)) , xysingvec=xysingvec(:m+p,:) )
```

Examples:

ex1_comp_mca2.F90

comp_mca_miss()

Purpose:

comp_mca_miss() performs a Maximum Covariance Analysis (MCA) or canonical covariance analysis between two data matrices *X* and *Y*, possibly containing missing values [Bretherton_etal:1992] [Bjornsson_Venegas:1997].

comp_mca_miss() computes the Singular Value Decomposition (SVD) of an estimate of the m-by-n correlation (or covariance) matrix XYCOR between two data matrices X and Y, possibly containing missing values. This SVD is written

$$XYCOR = U * S * V^T$$

where S is a min(m,n)-by-min(m,n) diagonal matrix, U is a m-by-min(m,n) orthogonal matrix, and V is a n-by-min(m,n) orthogonal matrix. The diagonal elements of S are the singular values of XYCOR, they are real and non-negative. The columns of U and V are, respectively, the left and right singular vectors of the estimate of XYCOR.

The subroutine computes the basic univariate statistics and the correlation (or covariance) matrix with only one pass through the data and allows out-of-core processing for the computation of the correlation (or covariance) matrix at the user option.

The means and standard-deviations of `X` and `Y` are computed from all valid data. The correlation (or covariance) coefficients are based on these univariate statistics and on all valid pairs of observations. The singular vectors and singular values are computed from these bivariate statistics.

The routine returns the singular values, the left and, optionally, the right singular vectors of the estimate of the correlation (or covariance) matrix `XYCOR` between the two data matrices `X` and `Y`.

The singular values and singular vectors of the covariance or correlation matrix are computed with the *bd_cmp()*, *ortho_gen_bd()* (or *ortho_gen_q_bd()*) and *bd_svd()* subroutines in module *SVD_Procedures*.

Finally, **comp_mca_miss()** may be used in a call with no observations (e.g. with `size(X,3-DIMVARX) = 0`) in order to finish the computations with *LAST* = `true`, when the total number of observations is unknown at the beginning of the computations.

Synopsis:

```
call comp_mca_miss( x(:m,:n) , y(:p,:n) , first , last , xstat(:m,:4) ,
↪ ystat(:p,:4) , xycor(:m,:p,:4) , xymiss , failure , dimvarx=dimvarx␣
↪, dimvary=dimvary , cov=cov , sort=sort , maxiter=maxiter ,␣
↪xysingval=xysingval(:min(m,p)) , xysingvar=xysingvar(:min(m,p)) ,␣
↪ysingvec=ysingvec(:p,:min(m,p)) )
```

comp_mca_miss2()

Purpose:

comp_mca_miss2() performs a Maximum Covariance Analysis (MCA) or canonical covariance analysis between two data matrices *X* and *Y*, possibly containing missing values [Bretherton_etal:1992] [Bjornsson_Venegas:1997].

comp_mca_miss2() computes a partial Singular Value Decomposition (SVD) of an estimate of the `m`-by-`n` correlation (or covariance) matrix `XYCOR` between two data matrices `X` and `Y`, possibly containing missing values. This partial SVD is written

$$XYCOR \simeq U(:m,:k) * S(:k,:k) * V(:n,:k)^T$$

where `S` is a `k`-by-`k` diagonal matrix, `U` is a `m`-by-`k` orthogonal matrix, and `V` is a `n`-by-`k` orthogonal matrix. The diagonal elements of `S(:k,:k)` are the largest singular values of `XYCOR`, they are real and non-negative. The columns of `U(:m,:k)` and `V(:n,:k)` are, respectively, the associated left and right singular vectors of the estimate of `XYCOR`.

The subroutine computes the basic univariate statistics and the correlation (or covariance) matrix with only one pass through the data and allows out-of-core processing for the computation of the correlation (or covariance) matrix at the user option.

The means and standard-deviations of `X` and `Y` are computed from all valid data. The correlation (or covariance) coefficients are based on these univariate statistics and on all valid pairs of observations. The singular vectors and singular values are computed from these bivariate statistics.

The routine returns all the singular values and, optionally, selected left and right singular vectors of the estimate of the correlation (or covariance) matrix `XYCOR` between the two data matrices `X` and `Y`.

Thus, **comp_mca_miss2()** must be used instead of *comp_mca_miss()* if you are only interested in the first few singular triplets of the estimate of `XYCOR`, which explains the larger part of the covariance or correlation between the data matrices *X* and *Y*, and for the processing of huge datasets.

The singular values and, optionally, selected singular vectors of the covariance or correlation matrix are computed (by inverse iteration) with the *svd_cmp()* and *bd_inviter2()* subroutines in module *SVD_Procedures*.

Finally, **comp_mca_miss2()** may be used in a call with no observations (e.g. with `size(X,3-DIMVARX) = 0`) in order to finish the computations with *LAST* = `true`, when the total number of observations is unknown at the beginning of the computations.

Synopsis:

```
call comp_mca_miss2( x(:m,:n) , y(:p,:n) , first , last , xstat(:m,:4) ,
↪ ystatt(:p,:4) , xycor(:m,:p,:4) , xymiss , failure , dimvarx=dimvarx␣
↪, dimvary=dimvary , cov=cov , xysingval=xysingval(:min(m,p)) ,␣
↪maxiter=maxiter , ortho=ortho , xysingvar=xysingvar(:min(m,p)) ,␣
↪xysingvec=xysingvec(:m+p,:) )
```

comp_pc_mca()

Purpose:

comp_pc_mca() computes estimates of Singular Variables (SV) and correlation (or covariance) fields from a data matrix *X* and a set of singular vectors *XSINGVEC* derived from the MCA analysis of the data matrix `X` with another matrix `Y`.

The subroutine computes the Singular Variables and the correlation (or covariance) fields with only one pass through the data and allows out-of-core processing at the user option.

This subroutine may be used in a call with no observations (e.g. `size(X,3-DIMVAR) = size(XPC,1) = 0`) in order to finish the computations with *LAST* = `true`, when the total number of observations is unknown at the beginning of the computations.

Synopsis:

```
call comp_pc_mca( x(:m,:n) , xsingvec(:m,:o) , first , last , xpccor(:m,:o) ,␣
↪pccorp(:(o*(o+1))/2) , xpc(:n,:o) , xn , dimvar=dimvar , xmean=xmean(:m) ,␣
↪xstd=xstd(:m) , xpcvar=xpcvar(:o)  )
```

Examples:

ex1_comp_mca.F90

ex1_comp_mca2.F90

comp_pc()

Purpose:

comp_pc() estimates of Principal Components (PC) from a data matrix *X* and eigenvectors or singular vectors derived from EOF or MCA analysis.

The subroutine computes the Principal Components with only one pass through the data, by projecting the observations onto the eigenvectors or singular vectors of the correlation or covariance matrix.

Synopsis:

```
call comp_pc( x(:m,:n) , xeigvec(:m)    , xpc(:n)    , dimvar=dimvar ,␣
↪xmean=xmean(:m) , xstd=xstd(:m) , xsingval=xsingval     )
call comp_pc( x(:m,:n) , xeigvec(:m,:o) , xpc(:n,:o) , dimvar=dimvar ,␣
↪xmean=xmean(:m) , xstd=xstd(:m) , xsingval=xsingval(:o) )
```

comp_pc_miss()

Purpose:

comp_pc_miss() estimates of Principal Components (PC) from a data matrix *X* and eigenvectors or singular vectors derived from EOF or MCA analysis, when *X* contains missing values.

The subroutine computes the Principal Components with only one pass through the data, by regressing the observations with non-missing values onto the eigenvectors or singular vectors of the correlation or covariance matrix.

When missing values are present, the Principal Components estimated by **comp_pc_miss()** are not necessarily uncorrelated.

Synopsis:

```
call comp_pc_miss( x(:m,:n) , xeigvec(:m)     , xpc(:n)     , xmiss ,␣
↪dimvar=dimvar , xmean=xmean(:m) , xstd=xstd(:m) , xsingval=xsingval )
call comp_pc_miss( x(:m,:n) , xeigvec(:m,:o) , xpc(:n,:o) , xmiss ,␣
↪dimvar=dimvar , xmean=xmean(:m) , xstd=xstd(:m) , xsingval=xsingval(:o) ,
↪ tol=tol , min_norm=min_norm )
```

5.26 MODULE FFT_Procedures

Module *FFT_Procedures* exports routines for (Fast) Fourier Transform (FFT) computations.

A large part of the documentation of this module is adapted from the nice documentation of the GNU Scientific Library [gsl].

Before going to the FFT, let us first briefly recall the Discrete Fourier Transform (DFT) [Bloomfield:1976] [Oppenheim_Schafer:1999]. For a complex valued sequence `z(k)` with length `n`, its `n`-point DFT is defined as,

$$x(j) = \sum_{k=0}^{n-1} z(k) W_n^{jk}$$

for $0 \leq j \leq n-1$ and where $W_n^{jk} = \exp(-2\pi ijk/n)$ with $i = \sqrt{-1}$ and $\pi = 3.1415923565....$ The key properties of the DFT are based on the following elementary identity

$$\sum_{k=0}^{n-1} W_n^{jk} = n\delta(j)$$

for $0 \leq j \leq n-1$ and where $\delta(j)$ takes 1 when j = 0 and 0 otherwise. The naive evaluation of the discrete Fourier transform is thus a matrix-vector multiplication $W * z$, which takes $O(n^2)$ operations for a `n`-valued complex sequence [Bloomfield:1976] [Oppenheim_Schafer:1999].

FFTs are efficient algorithms for computing the DFT, which use cleaver divide-and-conquer strategies to factorize the matrix W into smaller sub-matrices, corresponding to the integer factors of the length n. If n can be factorized into a product of integers $f_1 f_2 \ldots f_m$ then the DFT can be computed in $O(n \sum f_i)$ operations. For a radix-`2` FFT, this gives an operation count of $O(n \log_2 n)$ [Bloomfield:1976] [Oppenheim_Schafer:1999].

The module *FFT_Procedures* exports general routines, which work for complex valued arrays of any length. FFT routines for real valued sequences are also provided, but for real arrays of even length only (or of any length, but for a pair of real valued sequences of the same size). Routines for the FFT of complex and real arrays of up to three dimensions are also included.

Finally, DFTs for real sequences of any length, based on the Goertzel method, are also available here [Goertzel:1958] [Oppenheim_Schafer:1999]. This method is competitive with the FFT for the DFT of short sequences only.

Depending on the shape of the complex valued array to be transformed, a radix-`2` decimation-in-time Cooley-Tukey algorithm [Cooley_etal:1969] [Oppenheim_Schafer:1999], Bailey's Four-Step FFT algorithm [Bailey:1990] or a CHIRP-Z transform [Monro_Branch:1977] are used/combined to compute the complex or real FFTs. The radix-`2` decimation-in-time algorithm works only for lengths which are a power of two, but combined with the two other methods, this gives FFTs for complex arrays of any length.

At the user level, the routines provided here offer two types of transforms for complex and real sequences: forwards and backwards. Our definition of the *forward Fourier transform*, $x = \mathrm{FFT}(z)$, is,

$$x(j) = \sum_{k=0}^{n-1} z(k) \exp(-2\pi ijk/n)$$

for $0 \leq j \leq n-1$ and our definition of the *backward-inverse Fourier transform*, $x = \text{IFFT}(z)$, is,

$$z(j) = \frac{1}{n} \sum_{k=0}^{n-1} x(k) \exp(2\pi i j k/n).$$

for $0 \leq j \leq n-1$. The factor of $1/n$ makes this transform a *true inverse*. For example, a call to *fft()* with *FORWARD* = `true` followed by a call to *fft()* with *FORWARD* = `false` should return the original complex data (within numerical errors).

The following fragment of code illustrates how easy it it to compute the FFT of a complex valued sequence with the STATPACK FFT routines:

```
use FFT_Procedures, only: init_fft, fft
...
integer(i4b), parameter :: n=300
...
real(stnd), dimension(n) :: vec, vect
...
!
call init_fft( n )                                  ! Initialize the fft computations
call real_fft( vec(:n), vect(:n), forward=true )    ! Perform a forward fft, output␣
↪argument vect(:n) contains the forward FFT of vec(:n)
!
call end_fft( )                                     ! Deallocate internal fft workspace
```

For physical applications, it is important to remember that the index appearing in the DFT does not correspond directly to a physical frequency. If the time-step of the DFT is Δ then the frequency-domain includes both positive and negative frequencies, ranging from $-1/(2\Delta)$ through 0 to $+1/(2\Delta)$.

In the STATPACK FFT routines the positive frequencies are stored from the beginning of the output array argument up to the middle, and the negative frequencies are stored backwards from the end of the array.

Here is a table, which shows the correspondence between the time-domain data z and the frequency-domain data x, used in the STATPACK FFT routines (note that the index runs from 0 to $n-1$ as in our definition of the DFT):

```
index     z                 x = FFT(z)

0         z(t = 0)          x(f = 0)
1         z(t = 1)          x(f = 1/(n Delta))
2         z(t = 2)          x(f = 2/(n Delta))
.         ........          ..................
n/2       z(t = n/2)        x(f = +1/(2 Delta),
                                  -1/(2 Delta))
.         ........          ..................
n-3       z(t = n-3)        x(f = -3/(n Delta))
n-2       z(t = n-2)        x(f = -2/(n Delta))
n-1       z(t = n-1)        x(f = -1/(n Delta))
```

When n is even the location $n/2$ contains the most positive and negative frequencies $(+1/(2\Delta), -1/(2\Delta))$ which are equivalent. If n is odd then general structure of the table above still applies, but $n/2$ does not appear. Remind, finally, that the indexing in the above table is shifted by one compared to the *classical* Fortran convention.

The routines for real valued sequences are similar to those for complex sequences. However, there is an important difference because the Fourier transform of a real sequence is a complex sequence with a special symmetry:

$$z(k) = z(n-k)^*$$

A sequence with this symmetry is called *conjugate-complex* or *half-complex*. This symmetry of the half-complex sequence implies that only half of the complex numbers need to be computed and stored. The remaining half can

be reconstructed using the half-complex symmetry property. This explains, for example, why the output complex argument *VECT* of the routine *real_fft()*, which computes the FFT of a n-element real sequence vec, has a size of size(vec)/2 + 1 for a real valued sequence of even length size(vec).

The STATPACK FFT routines for a real valued sequence of even length, compute and store only the coefficients of the positive frequency half of the full complex Fourier transform of the input real valued sequence (see the table above). From this output *half-complex* sequence, the full complex Fourier transform of the input real valued sequence vec of even length n can be easily computed as illustrated by the following portion of code:

```
use FFT_Procedures, only: init_fft, real_fft
...
integer(i4b), parameter :: n=300, nd2 = n/2   ! n is even
...
real(stnd),    dimension(200) :: vec
complex(stnd), dimension(200) :: vect
...
!
call init_fft( nd2 )                                    ! Initialize the real fft␣
↪computations
call real_fft( vec(:n), vect(:nd2+1), forward=true )  ! Performs the real fft,␣
↪argument vect(:nd2+1) is the output half-complex sequence
!
vect(n:nd2+2:-1) = conjg( vect(2:nd2) )                 ! Compute the full complex␣
↪Fourier transform of vec(:n) using the symmetry
!
call end_fft( )                                         ! Deallocate internal fft␣
↪workspace
```

Note also that routines, which compute directly the backward-inverse Fourier transform (which is a real sequence) of an half-complex sequence, are not provided in this version STATPACK. The generic *real_fft()* routine does provide a backward Fourier transform for a real valued sequence, x_k, based on the following formula:

$$z(j) = \frac{1}{n} \sum_{k=0}^{n-1} x(k) \exp(2\pi i j k / n).$$

But, the result is again a half-complex sequence and this is not the *real* backward-inverse Fourier transform of the half-complex sequence obtained by a call to *real_fft()* with *FORWARD* = true.

Please note that routines provided in this module apply only to real/complex data of kind **stnd**. The real/complex kind type **stnd** is defined in module *Select_Parameters*.

In order to use one of these routines, you must include an appropriate use FFT_Procedures or use Statpack statement in your Fortran program, like:

```
use FFT_Procedures, only: init_fft
```

or:

```
use Statpack, only: init_fft
```

Here is the list of the public routines exported by module *FFT_Procedures*:

init_fft()

Purpose:

init_fft() sets up constants, the Chirp functions and the Fourier transform of the Chirp functions for use by other STATPACK FFT routines, which compute the FFT for a complex (or real) valued array.

init_fft() is first called to establish and transform the Chirp functions and other constants. Then, STATPACK FFT routines can be called any number of times without the precalculated constants being destroyed; a further call to **init_fft()** will only be necessary if Fourier transforms for a new length (or shape) are required.

Synopsis:

```
call init_fft ( shap(:n), dim=dim )
call init_fft ( length1 )
call init_fft ( length1, length2 )
call init_fft ( length1, length2, length3 )
```

Examples:

ex1_fft.F90

fftxy()

Purpose:

Given two real valued sequences (arrays) of the same length (shape), *X* and *Y*, **fftxy()** returns the Fast Fourier Transforms (FFT) of these sequences (arrays) in the two complex valued sequences (arrays) *FFTX* and *FFTY*.

Real arrays of up to three dimensions can be FFT by **fftxy()**. For arrays of two or three dimensions, the FFTs can be performed on a specific section of the arrays.

Synopsis:

```
call fftxy( x(:n)       , y(:n)       , fftx(:n)       , ffty(:n)           ␣
↪)
call fftxy( x(:m,:n)    , y(:m,:n)    , fftx(:m,:n)    , ffty(:m,:n)        ␣
↪)
call fftxy( x(:m,:p,:n) , y(:m,:p,:n) , fftx(:m,:p,:n) , ffty(:m,:p,:n)     ␣
↪)
call fftxy( x(:m,:n)    , y(:m,:n)    , fftx(:m,:n)    , ffty(:m,:n)    , dim␣
↪)
call fftxy( x(:m,:p,:n) , y(:m,:p,:n) , fftx(:m,:p,:n) , ffty(:m,:p,:n) , dim␣
↪)
```

Examples:

ex1_fftxy.F90

fft()

Purpose:

fft() implements the Fast Fourier Transform (FFT) for a complex valued sequence (or array) *DAT* of general length (or shape).

Complex array of up to three dimensions can be FFT by **fft()**. For arrays of two or three dimensions, the FFTs can be performed on a specific section of the arrays.

Synopsis:

```
call fft( dat(:n)       , forward )
call fft( dat(:m,:n)    , forward )
call fft( dat(:m,:p,:n) , forward       )
call fft( dat(:m,:n)    , forward , dim )
call fft( dat(:m,:p,:n) , forward , dim )
```

Examples:

ex1_fft.F90

ex1_real_fft.F90

fft_row()

Purpose:

fft_row() implements the Fast Fourier Transform for a complex valued sequence *DAT* of general length or for the row-sequences of a complex matrix *DAT*.

Synopsis:

```
call fft_row( dat(:n)     , forward )
call fft_row( dat(:m,:n) , forward )
```

Examples:

ex1_fft_row.F90

real_fft()

Purpose:

real_fft() computes the Fast Fourier Transform (FFT) for a real valued sequence *VEC* of even length or the FFTs of the columns of the real matrix *MAT*, which must also be of even length.

Only, the half-complex sequence of the full complex FFT is computed and stored in arguments *VECT* or *MATT*.

Synopsis:

```
call real_fft( vec(:n)     , vect(:(n/2)+1)     , forward )
call real_fft( mat(:m,:n) , matt(:m,:(n/2)+1) , forward )
```

Examples:

ex1_real_fft.F90

real_fft_forward()

Purpose:

real_fft_forward() implements the forward Discrete Fourier Transform (DFT) for a real valued sequence *VEC* of general length or of the row (*DIM* = 2) or column (*DIM* = 1) vectors of the real matrix *MAT*.

Only, the parts of the DFTs corresponding to the positive frequencies (e.g. the half-complex sequences of the full complex FFTs) are computed and output in the arguments *VECR* and *VECI* or *MATR* and *MATI* (rowwise).

The forward DFT is computed using Goertzel method and may be of general length.

Synopsis:

```
call real_fft_forward( vec(:n)     , vecr(:(n/2)+1) , veci(:(n/2)+1)       )
call real_fft_forward( mat(:m,:n) , matr(:,:)      , mati(:,:)      , dim )
```

Examples:

ex1_real_fft_forward.F90

real_fft_backward()

Purpose:

real_fft_backward() computes the (real) backward Discrete Fourier Transform (DFT) for half-complex valued sequences stored in:

- the vector *VECR* (real part of the half-complex complex sequence) and *VECI* (imaginary part of the half-complex sequence). The resulting real DFT is stored in the real vector *VEC*;

or

- the matrices *MATR* (real part of the half-complex sequences stored rowwise) and *MATI* (imaginary part of the half-complex sequences stored rowwise). The resulting real DFTs are stored in the rows (*DIM* = 2) or the columns (*DIM* = 1) of the real matrix *MAT*.

The backward DFT is computed using Goertzel method and may be of general length.

Synopsis:

```
call real_fft_backward( vecr(:(n/2)+1) , veci(:(n/2)+1) , vec(:n)          )
call real_fft_backward( matr(:,:)      , mati(:,:)       , mat(:m,:n) , dim )
```

Examples:

ex1_real_fft_forward.F90

end_fft()

Purpose:

end_fft() deallocates the workspace and internal variables previously allocated by a call to *init_fft()*.

Synopsis:

```
call end_fft( )
```

Examples:

ex1_fft.F90

5.27 MODULE Time_Series_Procedures

Module *Time_Series_Procedures* exports subroutines and functions for time series analysis.

Routines included in this module can be used to smooth and decompose (multi-channel) time series x_i into the models:

$$x_i = t_i + r_i$$

or

$$x_i = s_i + t_i + r_i$$

where i refers to a time index and the t_i term is used to quantify the trend and low-frequency variations in the time series, the s_i term describes the harmonic component (e.g., diurnal or seasonal cycle) and its modulation through time and, finally, the r_i term contains the residual component.

All the terms are estimated through a sequence of applications of locally weighted regression or low-order polynomial (e.g., LOESS) to data windows whose length is chosen by the user [Cleveland:1979] [Cleveland_Devlin:1988] [Cleveland_etal:1990].

Also included, are easy-to-use procedures for extracting frequency-defined series components from (multi-channel) time series based on the Fourier decomposition, which views the signal as a linear combination of purely harmonic components, each having a time-invariant amplitude and a well-defined frequency [Bloomfield:1976] [Duchon:1979] [Iacobucci_Noullez:2005].

These frequency filters can be obtained:

- by windowing [Oppenheim_Schafer:1999]., which consists of convolving a specific window (such as a raised-cosine or Hamming/Hanning window) with the ideal rectangular filter response function in the frequency domain and using the FFT to transform from the time and frequency domains for the application of the windowed filter to the signal (see the *hwfilter()* routine for example);

- by operating only in the time domain and using a moving data window which is centered on i-th sample for extracting the desired frequency component at the x_i observation of the time series.

$$W_i^H = \{x_{i-H}, \ldots, x_i, \ldots, x_{i+H}\}$$

Here, H is a non-negative integer called the *window half-length*, which represents the number of samples before and after sample i. The total window length, which is also the number of filter coefficients to compute, is $K = 2H + 1$.

Routines are then provided to compute the symmetric linear filter coefficients with the user-desired properties (e.g., low-pass, band-pass or high-pass) in a first step [Bloomfield:1976] [Duchon:1979]. See the *lp_coef()*, *lp_coef2()*, *hp_coef()*, *hp_coef2()*, *bd_coef()* and *bd_coef2()* routines for more details. These symmetric linear filter coefficients can then be applied to the signal in the time (or frequency) domain at the user option for performing the symmetric filtering operation of the time series in a second step (see the *symlin_filter()* and *symlin_filter2()* routines for example).

Finally, a large set of routines for spectral and cross-spectral estimations based on the FFT and smoothing the periodogram of time series in a variety of ways are also provided, see [Bloomfield:1976], [Welch:1967] and [Cooley_etal:1970], as well as a large variety of procedures for testing the hypothesis that two or several independent time-series are realizations of the same stationary process based on statistic computed from spectral density estimates of the time series [Diggle:1990].

Please note that routines provided in this module apply only to real data of kind **stnd**. The real kind type **stnd** is defined in module *Select_Parameters*.

In order to use one of these routines, you must include an appropriate `use Time_Series_Procedures` or `use Statpack` statement in your Fortran program, like:

```
use Time_Series_Procedures, only: comp_smooth
```

or:

```
use Statpack, only: comp_smooth
```

Here is the list of the public routines exported by module *Time_Series_Procedures*:

comp_smooth()

Purpose:

comp_smooth() smooths a time series or a multichannel time series given in the argument *X*.

The smoothing is equivalent to the application of a moving average of approximately `(2 * smooth_factor) + 1`, where `smooth_factor` is specified with the help of the input *SMOOTH_FACTOR* argument.

For more details, see [Olagnon:1996].

Synopsis:

```
call comp_smooth( x(:)     , smooth_factor                 )
call comp_smooth( x(:,:)   , smooth_factor , dimvar=dimvar )
call comp_smooth( x(:,:,:) , smooth_factor                 )
```

comp_trend()

Purpose:

comp_trend() extracts a smoothed component from a time series or a multichannel time series using a LOESS smoother [Cleveland:1979] [Cleveland_Devlin:1988].

In the LOESS procedure, the analyzed (multi-channel) time series is decomposed into two terms:

$$x_i = t_i + r_i$$

where i refers to a time index and the t_i term is used to quantify the trend and low-frequency variations in the time series and the r_i term contains the residual component.

The trend t_i is estimated through a sequence of applications of locally weighted regression or low-order polynomial (e.g., a LOESS smoother) to data windows whose length is chosen by the user. More precisely, at each point (x_k, k) locally weighted regression is used to smooth the time series and find the trend t_k. t_k is the the value at (x_k, k) of a polynomial fit to the data using weighted least squares, where the weight for (x_i, i) is large if i is closed to k and small if it is not.

The LOESS smoother for estimating the trend is specified with three parameters: a width (e.g., argument *NT*), a degree (e.g., argument *ITDEG*) and a jump (e.g., argument *NTJUMP*). The width specifies the number of data points that the local interpolation uses to smooth each point in the time series, the degree specifies the degree of the local polynomial that is fit to the data, and the jump specifies how many points are skipped between LOESS interpolations, with linear interpolation being done between these points.

If the optional *ROBUST* argument is set to `true`, the process is iterative and includes robustness iterations that take advantages of the weighted-least-squares underpinnings of LOESS to remove the effects of outliers [Cleveland:1979] [Cleveland_Devlin:1988].

comp_trend() returns the smoothed component (e.g., the trend) and, optionally, the robustness weights.

The input argument *Y* can be a time series (e.g., a vector) or a multichannel time series (e.g., a matrix and each column is a time series).

This subroutine is adapted from subroutine STL (Seasonal-Trend decomposition based on LOESS) developed by Cleveland and coworkers at AT&T Bell Laboratories [Cleveland_etal:1990]. But, **comp_trend()** assumes that the time series has no seasonal cycle or other harmonic components. If your time series include a seasonal cycle or other harmonic components, you must use *comp_stl()* or *comp_stlez()* instead.

Note, finally, that **comp_trend()** expects equally spaced data with no missing values.

Synopsis:

```
call comp_trend( y(:n)     , nt , itdeg , robust , trend(:n)     ,␣
↪ntjump=ntjump , maxiter=maxiter , rw=rw , no=no , ok=ok )
call comp_trend( y(:n,:p) , nt , itdeg , robust , trend(:n,:p) ,␣
↪ntjump=ntjump , maxiter=maxiter , rw=rw , no=no , ok=ok )
```

comp_stlez()

Purpose:

comp_stlez() decomposes a time series vector or the (time series) columns of a matrix into seasonal and trend components using a Seasonal-Trend decomposition based on LOESS (STL) [Cleveland_etal:1990]. In the STL procedure, the analyzed (multi-channel) time series is decomposed into three terms:

$$x_i = s_i + t_i + r_i$$

where i refers to a time index and the t_i term is used to quantify the trend and low-frequency variations in the time series, the s_i term describes the harmonic component (e.g., diurnal or seasonal cycle) and its modulation through time and, finally, the r_i term contains the residual component.

All the terms are estimated through a sequence of applications of locally weighted regression or low-order polynomial (e.g., LOESS) to data windows whose length is chosen by the user [Cleveland:1979] [Cleveland_Devlin:1988]. This process is iterative with many steps and may include robustness iterations (when the argument *ROBUST* is set to `true`) that take advantage of the weighted-least-squares underpinnings of LOESS to remove the effects of outliers [Cleveland_etal:1990].

There are three LOESS smoothers in the process and each require three parameters: a width, a degree, and a jump. The width specifies the number of data points that the local interpolation uses to smooth each point, the degree specifies

the degree of the local polynomial that is fit to the data, and the jump specifies how many points are skipped between LOESS interpolations, with linear interpolation being done between these points.

The LOESS smoother for estimating the trend is specified with the following parameters: a width (e.g., *NT*), a degree (e.g., *ITDEG*) and a jump (e.g., *NTJUMP*).

The LOESS smoother for estimating the seasonal component is specified with the following parameters: a width (e.g., *NS*), a degree (e.g., *ISDEG*) and a jump (e.g., *NSJUMP*).

The LOESS smoother for low-pass filtering is specified with the following parameters: a width (e.g., *NL*), a degree (e.g., *ILDEG*) and a jump (e.g., *NLJUMP*).

comp_stlez() is an iterative process, which may be interpreted as a frequency filter directly applicable to non-stationary (uni-dimensional) time series including harmonic components [Cleveland_etal:1990].

It returns the components and, optionally, the robustness weights.

This subroutine is a FORTRAN 90 implementation of subroutine STLEZ developed by Cleveland and coworkers at AT&T Bell Laboratories [Cleveland_etal:1990].

comp_stlez() offers an easy to use version of *comp_stl()* subroutine, also included in STATPACK, by defaulting most parameters values associated with the three LOESS smoothers described above and also used in *comp_stl()*.

At a minimum, **comp_stlez()** requires specifying:

- the periodicity of the data (e.g., the *NP* argument, `12` for monthly),
- the width of the LOESS smoother used to smooth the cyclic seasonal sub-series (e.g., the *NS* argument),
- the degree of the locally-fitted polynomial in seasonal smoothing (e.g., *ISDEG* argument),
- the degree of the locally-fitted polynomial in trend smoothing (e.g., *ITDEG* argument).

comp_stlez() sets, by default, others parameters of the STL procedure to the values recommended in [Cleveland_etal:1990] . It also includes tests of convergence if robust iterations are carried out. Otherwise, **comp_stlez()** is similar to *comp_stl()*.

If your time series do not include a seasonal cycle or other harmonic components, you must use *comp_trend()* instead of **comp_stlez()**.

Note, finally, that **comp_stlez()** expects equally spaced data with no missing values.

Synopsis:

```
call comp_stlez( y(:n)     , np , ns , isdeg , itdeg , robust , season(:n) ␣
↪  , trend(:n)     , ni=ni , nt=nt , nl=nl , ildeg=ildeg , nsjump=nsjump ,␣
↪ntjump=ntjump , nljump=nljump , maxiter=maxiter , rw=rw , no=no , ok=ok )
call comp_stlez( y(:n,:p) , np , ns , isdeg , itdeg , robust , season(:n,
↪:p) , trend(:n,:p) , ni=ni , nt=nt , nl=nl , ildeg=ildeg , nsjump=nsjump␣
↪, ntjump=ntjump , nljump=nljump , maxiter=maxiter , rw=rw , no=no , ok=ok )
```

comp_stl()

Purpose:

comp_stl() decomposes a time series vector or the (time series) columns of a matrix into seasonal and trend components using a Seasonal-Trend decomposition based on LOESS (STL) [Cleveland_etal:1990]. In the STL procedure, the analyzed (multi-channel) time series is decomposed into three terms:

$$x_i = s_i + t_i + r_i$$

where i refers to a time index and the t_i term is used to quantify the trend and low-frequency variations in the time series, the s_i term describes the harmonic component (e.g., diurnal or seasonal cycle) and its modulation through time and, finally, the r_i term contains the residual component.

All the terms are estimated through a sequence of applications of locally weighted regression or low-order polynomial (e.g., LOESS) to data windows whose length is chosen by the user [Cleveland:1979] [Cleveland_Devlin:1988]. This process is iterative with many steps and may include robustness iterations (when the argument *NO* is set to an integer value greater than `0`) that take advantage of the weighted-least-squares underpinnings of LOESS to remove the effects of outliers [Cleveland_etal:1990].

There are three LOESS smoothers in the process and each require three parameters: a width, a degree, and a jump. The width specifies the number of data points that the local interpolation uses to smooth each point, the degree specifies the degree of the local polynomial that is fit to the data, and the jump specifies how many points are skipped between LOESS interpolations, with linear interpolation being done between these points.

The LOESS smoother for estimating the trend is specified with the following parameters: a width (e.g., *NT*), a degree (e.g., *ITDEG*) and a jump (e.g., *NTJUMP*).

The LOESS smoother for estimating the seasonal component is specified with the following parameters: a width (e.g., *NS*), a degree (e.g., *ISDEG*) and a jump (e.g., *NSJUMP*).

The LOESS smoother for low-pass filtering is specified with the following parameters: a width (e.g., *NL*), a degree (e.g., *ILDEG*) and a jump (e.g., *NLJUMP*).

comp_stl() is an iterative process, which may be interpreted as a frequency filter directly applicable to non-stationary (uni-dimensional) time series including harmonic components [Cleveland_etal:1990].

It returns the components and, optionally, the robustness weights.

This subroutine is a FORTRAN 90 implementation of subroutine STL developed by Cleveland and coworkers at AT&T Bell Laboratories [Cleveland_etal:1990].

If your time series do not include a seasonal cycle or other harmonic components, you must use *comp_trend()* instead of **comp_stl()**. Also, if you don't know how or want to specify all the parameters in **comp_stl()**, you can use *comp_stlez()*, which is an easy to use version of **comp_stl()**.

Note, finally, that **comp_stl()** expects equally spaced data with no missing values.

Synopsis:

```
call comp_stl( y(:n)    , np , ni , no , isdeg , itdeg , ildeg , nsjump ,␣
↪ntjump , nljump , ns , nt , nl , rw , season(:n)    , trend(:n)    )
call comp_stl( y(:n,:p) , np , ni , no , isdeg , itdeg , ildeg , nsjump ,␣
↪ntjump , nljump , ns , nt , nl , rw , season(:n,:p) , trend(:n,:p) )
```

ma()

Purpose:

ma() smooths the vector *X* with a moving average of length *LEN* and output the result in the vector *AVE*.

This subroutine is a low-level subroutine used by subroutines *comp_stlez()* and *comp_stl()*.

Synopsis:

```
call ma( x(:n) , len , ave(:n) )
```

detrend()

Purpose:

detrend() detrends a time series (e.g., the argument *VEC*) or a multi-channel time series (e.g., the rows of the matrix argument *MAT*).

If:

- *TREND* = `1` The mean of the time series is removed
- *TREND* = `2` The drifts from the time series are estimated and removed by using the formula (for a time series):

```
slope = (VEC(size(VEC)) - VEC(1))/(size(VEC) - 1)
```

or (for a multi-channel time series):

```
slope(:)  = (MAT(:,size(MAT,2)) - MAT(:,1))/(size(MAT,2) - 1)
```

- *TREND* = 3 The least-squares lines from the time series are removed.

On exit, the original time series may be recovered with the formula (for a time series):

```
VEC(i) = VEC(i) + ORIG + SLOPE * real(i-1,stnd)
```

for i = 1, size(vec), or (for a multi-channel time series):

```
MAT(j,i) = MAT(j,i) + ORIG(j) + SLOPE(j) * real(i-1,stnd)
```

for i = 1, size(MAT,2) and j = 1, size(MAT,1), in all the cases.

Synopsis:

```
call detrend( vec(:n)    , trend , orig=orig ,     slope=slope     )
call detrend( vec(:p,:n) , trend , orig=orig(:p) , slope=slope(:p) )
```

hwfilter()

Purpose:

hwfilter() filters a time series (e.g., the vector argument *VEC*) or a multi-channel time series (e.g., the columns of the matrix argument *MAT*) in the frequency band limited by periods *PL* and *PH* by Hamming/Hanning-windowed (HW) filtering and a Fast Fourier Transform algorithm [Iacobucci_Noullez:2005].

PL and *PH* are expressed in number of points, i.e. *PL* = 6(18) and *PH* = 32(96) selects periods between 1.5 yrs and 8 yrs for quarterly (monthly) data, as an illustration.

Use *PL* = 0 for high-pass filtering frequencies corresponding to periods shorter than *PH*, or *PH* = 0 for low-pass filtering frequencies corresponding to periods longer than *PL*.

Setting *PH* < *PL* is also allowed and performs band rejection of periods between *PH* and *PL* (i.e. in that case the meaning of the *PL* and *PH* arguments are reversed).

The frequency filter implemented in **hwfilter()** is obtained by convolving a raised-cosine window with the ideal rectangular filter response function. This windowed filter has almost no leakage and has a very flat response in the pass-band. Moreover, this filter is stationary and symmetric and, therefore, it induces no phase-shift. It is thus a good filter for extracting frequency-defined series components for short-length time series.

For more details, see [Iacobucci_Noullez:2005].

Synopsis:

```
call hwfilter( vec(:)   , pl , ph , initfft=initfft , trend=trend , win=win
↪                     )
call hwfilter( mat(:,:) , pl , ph , initfft=initfft , trend=trend , win=win ,
↪max_alloc=max_alloc )
```

Examples:

ex1_hwfilter.F90

ex2_hwfilter.F90

hwfilter2()

Purpose:

hwfilter2() filters a time series (e.g., the vector argument *VEC*) or a multi-channel time series (e.g., the columns of the matrix argument *MAT*) in the frequency band limited by periods *PL* and *PH* by Hamming/Hanning-windowed (HW) filtering [Iacobucci_Noullez:2005].

PL and *PH* are expressed in number of points, i.e. *PL* = `6(18)` and *PH* = `32(96)` selects periods between `1.5` yrs and `8` yrs for quarterly (monthly) data, as an illustration.

Use *PL* = `0` for high-pass filtering frequencies corresponding to periods shorter than *PH*, or *PH* = `0` for low-pass filtering frequencies corresponding to periods longer than *PL*.

Setting *PH* < *PL* is also allowed and performs band rejection of periods between *PH* and *PL* (i.e. in that case the meaning of the *PL* and *PH* arguments are reversed).

The frequency filter implemented in **hwfilter2()** is obtained by convolving a raised-cosine window with the ideal rectangular filter response function. This windowed filter has almost no leakage and has a very flat response in the pass-band. Moreover, this filter is stationary and symmetric and, therefore, it induces no phase-shift. It is thus a good filter for extracting frequency-defined series components for short-length time series.

The unique difference between **hwfilter2()** and *`hwfilter()`* is the use of the Goertzel method for computing the Fourier transform of the data (as in [Iacobucci_Noullez:2005]) instead of a Fast Fourier Transform algorithm.

For more details, see [Iacobucci_Noullez:2005].

Synopsis:

```
call hwfilter2( vec(:)   , pl , ph , trend=trend , win=win )
call hwfilter2( mat(:,:) , pl , ph , trend=trend , win=win )
```

Examples:

ex1_hwfilter2.F90

ex2_hwfilter2.F90

`lp_coef()`

Purpose:

lp_coef() computes the *K*-term least squares approximation to an -ideal- low pass filter with cutoff period *PL* (e.g., cutoff frequency $FC = 1/PL$).

This filter has a transfer function with a transition band of width `delta` surrounding *FC* equals to

$$delta = 4 * \pi / K$$

when *FC* is expressed in radians.

lp_coef() computes symmetric linear low-pass filter coefficients using a least squares approximation to an ideal low-pass filter with convergence factors (i.e. a Lanczos window) which reduce overshoot and ripple [Bloomfield:1976].

This low-pass filter has a transfer function which changes from approximately one to zero in a transition band about the ideal cutoff frequency *FC* ($FC = 1/PL$), that is from $(FC - 1/K)$ to $(FC + 1/K)$, as discussed in section 6.4 of [Bloomfield:1976].

The user must specify the cutoff period (or the cutoff frequency) and the number of filter coefficients, which must be odd.

The user must also choose the number of filter coefficients, *K*, so that $(FC - 1/K) \geq 0$ and $(FC + 1/K) < 0.5$ if the optional logical argument *NOTEST_FC* is not used or is not set to `true`.

In addition, *K* must be chosen as a compromise between:

- a sharp cutoff, that is, $1/K$ small;
- and minimizing the number of data points lost by the filtering operations (e.g., $(K - 1)/2$ data points will be lost from each end of the series).

The subroutine returns the normalized low-pass filter coefficients.

For more details and algorithm, see Chapter 6 of [Bloomfield:1976].

Synopsis:

```
coef(:k) = lp_coef( pl , k , fc=fc , notest_fc=notest_fc )
```

Examples:

ex1_lp_coef.F90

lp_coef2()

Purpose:

lp_coef2() computes the K-term least squares approximation to an -ideal- low pass filter with cutoff period *PL* (e.g., cutoff frequency $FC = 1/PL$) by windowed filtering (e.g., Hamming window is used).

This filter has a transfer function with a transition band of width `delta` surrounding *FC* equals to

$$delta = 4 * \pi / K$$

when *FC* is expressed in radians.

lp_coef2() computes symmetric linear low-pass filter coefficients using a least squares approximation to an ideal low-pass filter. The Hamming window is used to reduce overshoot and ripple in the transfer function of the ideal low-pass filter [Bloomfield:1976].

This low-pass filter has a transfer function which changes from approximately one to zero in a transition band about the ideal cutoff frequency *FC* ($FC = 1/PL$), that is from $(FC - 1/K)$ to $(FC + 1/K)$, as discussed in section 6.4 of [Bloomfield:1976].

The user must specify the cutoff period (or the cutoff frequency) and the number of filter coefficients, which must be odd.

The user must also choose the number of filter coefficients, *K*, so that $(FC - 1/K) \geq 0$ and $(FC + 1/K) < 0.5$ if the optional logical argument *NOTEST_FC* is not used or is not set to `true`.

The overshoot and the associated ripples in the ideal transfer function are reduced by the use of the Hamming window.

In addition, *K* must be chosen as a compromise between:

- a sharp cutoff, that is, $1/K$ small;
- and minimizing the number of data points lost by the filtering operations (e.g., $(K - 1)/2$ data points will be lost from each end of the series).

The subroutine returns the normalized low-pass filter coefficients.

For more details and algorithm, see Chapter 6 of [Bloomfield:1976].

Synopsis:

```
coef(:k) = lp_coef2( pl , k , fc=fc , win=win , notest_fc=notest_fc )
```

Examples:

ex1_lp_coef2.F90

hp_coef()

Purpose:

hp_coef() computes the K-term least squares approximation to an -ideal- high pass filter with cutoff period *PH* (e.g., cutoff frequency $FC = 1/PH$).

This filter has a transfer function with a transition band of width `delta` surrounding *FC* equals to

$$delta = 4 * \pi / K$$

when *FC* is expressed in radians.

hp_coef() computes symmetric linear high-pass filter coefficients from the corresponding low-pass filter as given by function *lp_coef()*. This is equivalent to subtracting the low-pass filtered series from the original time series.

This high-pass filter has a transfer function which changes from approximately zero to one in a transition band about the ideal cutoff frequency *FC* ($FC = 1/PH$), that is from $(FC - 1/K)$ to $(FC + 1/K)$, as discussed in section 6.4 of [Bloomfield:1976].

The user must specify the cutoff period (or the cutoff frequency) and the number of filter coefficients, which must be odd.

The user must also choose the number of filter coefficients, *K*, so that $(FC - 1/K) \geq 0$ and $(FC + 1/K) < 0.5$ if the optional logical argument *NOTEST_FC* is not used or is not set to `true`.

In addition, *K* must be chosen as a compromise between:

- a sharp cutoff, that is, $1/K$ small;
- and minimizing the number of data points lost by the filtering operations (e.g., $(K - 1)/2$ data points will be lost from each end of the series).

The subroutine returns the high-pass filter coefficients.

For more details and algorithm, see Chapter 6 of [Bloomfield:1976].

Synopsis:

```
coef(:k) = hp_coef( ph , k , fc=fc , notest_fc=notest_fc )
```

Examples:

ex1_hp_coef.F90

hp_coef2()

Purpose:

hp_coef2() computes the *K*-term least squares approximation to an -ideal- high pass filter with cutoff period *PH* (e.g., cutoff frequency $FC = 1/PH$) by windowed filtering (e.g., Hamming window is used).

This filter has a transfer function with a transition band of width `delta` surrounding *FC* equals to

$$delta = 4 * \pi/K$$

when *FC* is expressed in radians.

hp_coef() computes symmetric linear high-pass filter coefficients from the corresponding low-pass filter as given by function *lp_coef2()*. This is equivalent to subtracting the low-pass filtered series from the original time series.

This high-pass filter has a transfer function which changes from approximately zero to one in a transition band about the ideal cutoff frequency *FC* ($FC = 1/PH$), that is from $(FC - 1/K)$ to $(FC + 1/K)$, as discussed in section 6.4 of [Bloomfield:1976].

The user must specify the cutoff period (or the cutoff frequency) and the number of filter coefficients, which must be odd.

The user must also choose the number of filter coefficients, *K*, so that $(FC - 1/K) \geq 0$ and $(FC + 1/K) < 0.5$ if the optional logical argument *NOTEST_FC* is not used or is not set to `true`.

The overshoot and the associated ripples in the ideal transfer function are reduced by the use of the Hamming window.

In addition, *K* must be chosen as a compromise between:

- a sharp cutoff, that is, $1/K$ small;

- and minimizing the number of data points lost by the filtering operations (e.g., $(K-1)/2$ data points will be lost from each end of the series).

The subroutine returns the high-pass filter coefficients.

For more details and algorithm, see Chapter 6 of [Bloomfield:1976].

Synopsis:

```
coef(:k) = hp_coef2( ph , k , fc=fc , win=win , notest_fc=notest_fc )
```

Examples:

ex1_hp_coef2.F90

bd_coef()

Purpose:

bd_coef() computes the *K*-term least squares approximation to an -ideal- band pass filter with cutoff periods *PL* and *PH* (e.g., cutoff frequencies $1/PL$ and $1/PH$, respectively).

PL and *PH* are expressed in number of points, i.e. *PL* = `6(18)` and *PH* = `32(96)` selects periods between `1.5` yrs and `8` yrs for quarterly (monthly) data, as an illustration.

Alternatively, the user can directly specify the two cutoff frequencies, *FCL* and *FCH*, corresponding to *PL* and *PH*.

bd_coef() computes symmetric linear band-pass filter coefficients using a least squares approximation to an ideal band-pass filter that has convergence factors which reduce overshoot and ripple [Bloomfield:1976].

This band-pass filter is computed as the difference between two low-pass filters with cutoff frequencies $1/PH$ and $1/PL$, respectively (or *FCH* and *FCL*).

This band-pass filter has a transfer function which changes from approximately zero to one and one to zero in the transition bands about the ideal cutoff frequencies $1/PH$ and $1/PL$), that is from $(1/PH - 1/K)$ to $(1/PH + 1/K)$ and $(1/PL - 1/K)$ to $(1/PL + 1/K)$, respectively.

The user must specify the two cutoff periods and the number of filter coefficients, which must be odd.

The user must also choose the number of filter terms, *K*, so that:

- $0 \leq (1/PH - 1/K)$
- $(1/PH + 1.3/(K+1)) \leq (1/PL - 1.3/(K+1))$
- $(1/PL + 1/K) < 0.5$

However, if the optional logical argument *NOTEST_FC* is used and is set to `true`, the two tests

- $0 \leq (1/PH - 1/K)$
- $(1/PL + 1/K) < 0.5$

are bypassed.

In addition, *K* must be chosen as a compromise between:

- a sharp cutoff, that is, $1/K$ small;
- and minimizing the number of data points lost by the filtering operations (e.g., $(K-1)/2$ data points will be lost from each end of the series).

The subroutine returns the difference between the two corresponding normalized low-pass filter coefficients as computed by function *lp_coef()*.

For more details and algorithm, see Chapter 6 of [Bloomfield:1976] and [Duchon:1979].

Synopsis:

```
coef(:k) = bd_coef( pl , ph , k , fch=fch , fcl=fcl , notest_fc=notest_fc )
```

Examples:

ex1_bd_coef.F90

bd_coef2()

Purpose:

bd_coef2() computes the *K*-term least squares approximation to an -ideal- band pass filter with cutoff periods *PL* and *PH* (e.g., cutoff frequencies $1/PL$ and $1/PH$, respectively) by windowed filtering (e.g., Hamming window is used) [Bloomfield:1976].

PL and *PH* are expressed in number of points, i.e. *PL* = `6(18)` and *PH* = `32(96)` selects periods between `1.5` yrs and `8` yrs for quarterly (monthly) data, as an illustration.

Alternatively, the user can directly specify the two cutoff frequencies, *FCL* and *FCH*, corresponding to *PL* and *PH*.

bd_coef2() computes symmetric linear band-pass filter coefficients using a least squares approximation to an ideal band-pass filter. The Hamming window is used to reduce overshoot and ripple in the transfer function of the ideal low-pass filter.

This band-pass filter is computed as the difference between two low-pass filters with cutoff frequencies $1/PH$ and $1/PL$, respectively (or *FCH* and *FCL*).

This band-pass filter has a transfer function which changes from approximately zero to one and one to zero in the transition bands about the ideal cutoff frequencies $1/PH$ and $1/PL$), that is from $(1/PH - 1/K)$ to $(1/PH + 1/K)$ and $(1/PL - 1/K)$ to $(1/PL + 1/K)$, respectively.

The user must specify the two cutoff periods and the number of filter coefficients, which must be odd.

The user must also choose the number of filter terms, *K*, so that:

- $0 \leq (1/PH - 1/K)$
- $1/PH < 1/PL$
- $(1/PL + 1/K) < 0.5$

However, if the optional logical argument *NOTEST_FC* is used and is set to `true`, the two tests

- $0 \leq (1/PH - 1/K)$
- $(1/PL + 1/K) < 0.5$

are bypassed.

The overshoot and the associated ripples in the ideal transfer function are reduced by the use of the Hamming window.

In addition, *K* must be chosen as a compromise between:

- a sharp cutoff, that is, $1/K$ small;
- and minimizing the number of data points lost by the filtering operations (e.g., $(K - 1)/2$ data points will be lost from each end of the series).

The subroutine returns the difference between the two corresponding normalized low-pass filter coefficients as computed by function *lp_coef2()*.

For more details and algorithm, see Chapter 6 of [Bloomfield:1976].

Synopsis:

```
coef(:k) = bd_coef2( pl , ph , k , fch=fch , fcl=fcl , win=win , notest_
↪fc=notest_fc )
```

Examples:

ex1_bd_coef2.F90

pk_coef()

Purpose:

pk_coef() computes the *K*-term least squares approximation to an -ideal- band pass filter with peak response near one at the single frequency *FREQ* (e.g., the peak response is at $period = 1/FREQ$).

pk_coef() computes symmetric linear band-pass filter coefficients using a least squares approximation to an ideal band-pass filter that has convergence factors which reduce overshoot and ripple [Bloomfield:1976].

This band-pass filter is computed as the difference between two low-pass filters with cutoff frequencies `FCL` and `FCH`, respectively. See [Duchon:1979] for the computations of the two cutoff frequencies `FCL` and `FCH`.

This band-pass filter has a transfer function which changes from approximately zero to one and one to zero in the transition bands about the cutoff frequencies `FCH` and `FCL`, that is from $(FCH - 1/K)$ to *FREQ* and *FREQ* to $(FCL + 1/K)$, respectively.

The user must specify the frequency *FREQ* with unit response and the number of filter coefficients, *K*, which must be odd. The user must also choose the number of filter terms, *K*, as a compromise between:

- a sharp cutoff, that is, $1/K$ small;
- and minimizing the number of data points lost by the filtering operations (e.g., $(K - 1)/2$ data points will be lost from each end of the series).

The subroutine returns the difference between the two corresponding normalized low-pass filter coefficients as computed by function *lp_coef()*.

For more details and algorithm, see Chapter 6 of [Bloomfield:1976] and [Duchon:1979].

Synopsis:

```
coef(:k) = pk_coef( freq , k , notest_freq=notest_freq )
```

Examples:

ex1_pk_coef.F90

moddan_coef()

Purpose:

moddan_coef() computes the impulse response function (e.g., weights) corresponding to a number of applications of modified Daniell filters as done in subroutine *moddan_filter()*.

For definition, more details and algorithm, see [Bloomfield:1976].

Synopsis:

```
coef(:k) = moddan_coef( k , smooth_param(:) )
```

freq_func()

Purpose:

freq_func() computes the frequency response function (e.g., the transfer function) of the symmetric linear filter given by the argument *COEF(:)*.

The frequency response function is computed at *NFREQ* frequencies regularly sampled between zero and the Nyquist frequency if the optional logical argument *FOUR_FREQ* is not used or at the *NFREQ* Fourier frequencies $2 * \pi * j/nfreq$ for j = 0 to $nfreq - 1$ if this argument is used and set to `true`.

For more details, see [Bloomfield:1976] and [Oppenheim_Schafer:1999].

Synopsis:

```
call freq_func( nfreq , coef(:) , freqr(:nfreq) , four_freq=four_freq ,␣
↪freq=freq(:nfreq) )
```

Examples:

ex1_freq_func.F90

ex1_pk_coef.F90

symlin_filter()

Purpose:

symlin_filter() performs a symmetric filtering operation on an input time series (e.g., the vector argument *VEC*) or multi-channel time series (e.g., the matrix argument *MAT*).

The filtering is done in place and `(size(COEF)-1)/2` observations will be lost from each end of the (multi-channel) time series.

Note, also, that the filtered (multi-channel) time series is shifted in time and is stored on output in:

- `VEC(1:NFILT)`, with *NFILT* = `size(VEC) - size(COEF) + 1`.
- `MAT(:,1:NFILT)`, with *NFILT* = `size(MAT,2) - size(COEF) + 1`.

The symmetric linear filter coefficients (e.g., the array *COEF*) can be computed with the help of functions `lp_coef`, `lp_coef2`, `hp_coef`, `hp_coef2`, `bd_coef` and `bd_coef2`.

Synopsis:

```
call symlin_filter( vec(:)   , coef(:) , trend=trend , nfilt=nfilt )
call symlin_filter( mat(:,:) , coef(:) , trend=trend , nfilt=nfilt )
```

Examples:

ex1_symlin_filter.F90

ex1_bd_coef.F90

symlin_filter2()

Purpose:

symlin_filter2() performs a symmetric filtering operation on an input time series (e.g., the vector argument *VEC*) or multi-channel time series (e.g., the matrix argument *MAT*).

No time observations will be lost, however the first and last `(size(COEF)-1)/2` time observations are affected by end effects.

If *USEFFT* is used with the value `true`, the values at both ends of the output (multi-channel) series are computed by assuming that the input (multi-channel) series is part of a periodic sequence of period `size(VEC)` (or `size(MAT, 2)`. Otherwise, each end of the filtered (multi-channel)time series is estimated by truncated the symmetric linear filter coefficients array *COEF(:)*.

The symmetric linear filter coefficients (e.g., the array *COEF*) can be computed with the help of functions `lp_coef`, `lp_coef2`, `hp_coef`, `hp_coef2`, `bd_coef` and `bd_coef2`.

Synopsis:

```
call symlin_filter2( vec(:)   , coef(:) , trend=trend , usefft=usefft ,␣
↪initfft=initfft )
call symlin_filter2( mat(:,:) , coef(:) , trend=trend , usefft=usefft ,␣
↪initfft=initfft )
```

Examples:

ex1_symlin_filter2.F90

dan_filter()

Purpose:

dan_filter() smooths an input time series (e.g., the vector argument *VEC*) or multi-channel time series (e.g., the matrix argument *MAT*) by applying a Daniell filter (e.g., a simple moving average) of length *NSMOOTH*.

dan_filter() smooths an input (multi-channel) time series by applying a Daniell filter as discussed in chapter 7 of [Bloomfield:1976].

This subroutine use the hypothesis of an (even or odd) symmetry of the input (multi-channel) time series to avoid losing values from the ends of the series.

For more details and algorithm, see chapter 7 of [Bloomfield:1976].

Synopsis:

```
call dan_filter( vec(:)   , nsmooth , sym=sym , trend=trend )
call dan_filter( mat(:,:) , nsmooth , sym=sym , trend=trend )
```

moddan_filter()

Purpose:

moddan_filter() smooths an input time series (e.g., the vector argument *VEC*) or multi-channel time series (e.g., the matrix argument *MAT*) by applying a sequence of modified Daniell filters.

moddan_filter() smooths an input (multi-channel) time series by applying a sequence of modified Daniell filters as discussed in chapter 7 of [Bloomfield:1976]. This subroutine use the hypothesis of an (even or odd) symmetry of the input time series to avoid losing values from the ends of the series.

For more details and algorithm, see chapter 7 of [Bloomfield:1976].

Synopsis:

```
call moddan_filter( vec(:)   , smooth_param(:) , sym=sym , trend=trend )
call moddan_filter( mat(:,:) , smooth_param(:) , sym=sym , trend=trend )
```

extend()

Purpose:

extend() returns the *INDEX*-th term in the time series *VEC* or the multi-channel time series *MAT*, extending it if necessary with an even or odd symmetry according to the sign of *SYM*, which should be either plus or minus one. Note also that the value zero will result in the extended value being zero.

For more details and algorithm, see Chapter 6 of [Bloomfield:1976].

Synopsis:

```
x     = extend( vec(:p)    , index , sym )
x(:n) = extend( mat(:n,:p) , index , sym )
```

taper()

Purpose:

taper() applies a split-cosine-bell taper on an input time series *VEC* or a multi-channel time series *MAT*.

This subroutine is adapted from Chapter 5 of [Bloomfield:1976].

Synopsis:

```
call taper( vec(:)   , taperp )
call taper( mat(:,:) , taperp )
```

data_window()

Purpose:

data_window() computes data windows used in spectral computations.

For more details, see Chapter 5 of [Bloomfield:1976].

Synopsis:

```
wk(:n) = data_window( n  , win , taperp=taperp )
```

estim_dof()

Purpose:

estim_dof() computes *the equivalent number of degrees of freedom* of power and cross spectrum estimates as calculated by subroutines *power_spectrum()*, *cross_spectrum()*, *power_spectrum2()* and *cross_spectrum2()*.

The computed equivalent number of degrees of freedom must be divided by two for the zero and Nyquist frequencies.

Furthermore, the computed equivalent number of degrees of freedom is not right near the zero and Nyquist frequencies if the Power Spectral Density (PSD) estimates have been smoothed by modified Daniell filters.

The reason is that **estim_dof()** assumes that smoothing involves averaging independent frequency ordinates. This is true except near the zero and Nyquist frequencies where an average may contain contributions from negative frequencies, which are identical to and hence not independent of positive frequency spectral values. Thus, the number of degrees of freedom in PSD estimates near the zero and Nyquist frequencies are as little as half the number of degrees of freedom of the spectral estimates away from these frequency extremes if the optional argument *SMOOTH_PARAM* is used.

For more details and algorithm, see [Bloomfield:1976] and [Welch:1967].

Synopsis:

```
edof = estim_dof( wk(:n) , win=win , smooth_param=smooth_param , l0=l0 ,␣
↪nseg=nseg , overlap=overlap )
```

estim_dof2()

Purpose:

estim_dof2() computes *the equivalent number of degrees of freedom* of power and cross spectrum estimates as calculated by subroutines `power_spectrm()`, `cross_spectrm()`, `power_spectrm2()` and `cross_spectrm2()`.

For more details and algorithm, see [Bloomfield:1976] and [Welch:1967].

Synopsis:

```
edof(:(n+l0)/2 + 1 ) = estim_dof2( wk(:n) , l0 , win=win , nsmooth=nsmooth ,␣
↪nseg=nseg , overlap=overlap )
```

comp_conflim()

Purpose:

comp_conflim() estimates confidence limit factors for spectral estimates and, optionally, critical values for testing the null hypothesis that the squared coherencies between two time series are zero.

Synopsis:

```
call comp_conflim( edof     , probtest=probtest , conlwr=conlwr     ,␣
↪conupr=conupr     , testcoher=testcoher     )
call comp_conflim( edof(:n) , probtest=probtest , conlwr=conlwr(:n) ,␣
↪conupr=conupr(:n) , testcoher=testcoher(:n) )
```

spctrm_ratio()

Purpose:

spctrm_ratio() calculates a point-wise tolerance intervals for the ratios of two estimated spectra under the assumption that the two "true" underlying spectra are the same.

For more details, see Chapter 4 of [Diggle:1990].

Synopsis:

```
call spctrm_ratio( edofn     , edofd     , lwr_ratio     , upr_ratio     ,␣
↪pinterval=pinterval )
call spctrm_ratio( edofn(:n) , edofd(:n) , lwr_ratio(:n) , upr_ratio(:n) ,␣
↪pinterval=pinterval )
```

spctrm_ratio2()

Purpose:

spctrm_ratio2() calculates a conservative critical probability values (e.g., p-values) for testing the hypothesis of a common spectrum for two estimated (multi-channel) spectra (e.g., the arguments *PSVECN*, *PSVECD* or *PSMATN*, *PSMATD*).

These conservative critical probability values are computed from the minimum and maximum values of the ratio of the two estimated (multichannel) spectra and the associated probabilities of obtaining, respectively, a value less (for the minimum ratios) and higher (for the maximum ratios) than attained under the null hypothesis of a common spectra for the two (multichannel) time series.

This statistical test relies on the assumptions that the different spectral ordinates have the same sampling distribution and are independent of each other for each series. This means, in particular, that the spectral ordinates corresponding to the zero and Nyquist frequencies must be excluded from the *PSVECN* and *PSVECD* vectors (or *PSMATN*, *PSMATD* matrices) before calling **spctrm_ratio2**() and that the two estimated (multi-channel) spectra have not been obtained by smoothing the periodograms in the frequency domain, but by averaging different periodograms computed on replicated time series.

It is also assumed that the (multichannel) time series with spectra *PSVECN* and *PSVECD* (or *PSMATN* and *PSMATD*) are independent realizations.

For more details, see Chapter 4 of [Diggle:1990].

Synopsis:

```
call spctrm_ratio2( psvecn(:n)     , psvecd(:n)     , edofn , edofd , prob     ,
↪ min_ratio=min_ratio     , max_ratio=max_ratio     , prob_min_ratio=prob_min_
↪ratio     , prob_max_ratio=prob_max_ratio     )
call spctrm_ratio2( psmatn(:p,:n) , psmatd(:p,:n) , edofn , edofd , prob(:p)␣
↪, min_ratio=min_ratio(:p) , max_ratio=max_ratio(:p) , prob_min_ratio=prob_
↪min_ratio(:p) , prob_max_ratio=prob_max_ratio(:p) )
```

spctrm_ratio3()

Purpose:

spctrm_ratio3() calculates approximate critical probability values (e.g., p-values) for testing the hypothesis of a common spectrum for two estimated (multi-channel) spectra (e.g., the vector arguments *PSVECN*, *PSVECD* or matrix arguments *PSMATN*, *PSMATD*). These approximate critical probability values are derived from the following chi-squared log-ratio statistics:

- $chi2 = \frac{1}{(2/edofn)+(2/edofd)} \sum_{k=1}^{\nu} \ln(PSVECN(k)/PSVECD(k))^2$

  where ν = `size(PSVECN)` = `size(PSVECD)`

- $chi2(:n) = \frac{1}{(2/edofn)+(2/edofd)} \sum_{k=1}^{\nu} \ln(PSMATN(:n,k)/PSMATD(:n,k))^2$

  where ν = `size(PSMATN,2)` = `size(PSMATD,2)` and n = `size(PSMATN,1)` = `size(PSMATD,1)` is the number of channels in the two multi-channel time series.

In both cases, ν is the number of frequencies considered. Arguments *EDOFN* and *EDOFD* give, respectively, the equivalent numbers of degrees of freedom, $edofn$ and $edofd$, of the first and second estimated spectra (e.g., the numerator and denominator of the ratio of the two estimated spectra). These numbers can be computed by the *estim_dof()* and *estim_dof2()* functions.

In order to derive approximate critical probability values, it is assumed that $chi2$ (or $chi2(i)$ for i = 1 to n) has an approximate chi-squared distribution with ν degrees of freedom: $chi2 \sim \chi^2_\nu$ [Jenkins_Watts:1968] [Priestley:1981].

The chi-squared log-ratio statistics $chi2$ are stored on output in the *CHI2_STAT* scalar or vector arguments.

This statistical test relies on the assumptions that the different spectral ordinates have the same sampling distribution and are independent of each other for each time series. This means, in particular, that the spectral ordinates corresponding to the zero and Nyquist frequencies must be excluded from the *PSVECN* and *PSVECD* vector (or *PSMATN* and *PSMATD* matrix) spectra before calling **spctrm_ratio3()** and that the two estimated (multi-channel) spectra have not been obtained by smoothing the periodogram in the frequency domain, but by averaging different periodograms computed on replicated time series.

Thus, this test could only be used to compare two periodograms or two spectral estimates computed as the average of, say, r periodograms for each time series.

It is also assumed that the (multichannel) time series with spectra *PSVECN* and *PSVECD* (or *PSMATN* and *PSMATD*) are independent realizations.

Synopsis:

```
call spctrm_ratio3( psvecn(:n)     , psvecd(:n)     , edofn , edofd , chi2_stat␣
↪     , prob      )
call spctrm_ratio3( psmatn(:p,:n) , psmatd(:p,:n) , edofn , edofd , chi2_
↪stat(:p) , prob(:p) )
```

spctrm_ratio4()

Purpose:

spctrm_ratio4() calculates approximate critical probability values (e.g., p-values) for testing the hypothesis of a common shape for two estimated (multi-channel) spectra (e.g., the vector arguments *PSVECN*, *PSVECD* or matrix arguments *PSMATN*, *PSMATD*). These approximate critical probability values are derived from the following range log-ratio statistics:

- $range = \frac{1}{\sqrt{(2/edofn)+(2/edofd)}} (\max_{k=1}^{\nu} \ln(PSVECN(k)/PSVECD(k)) - \min_{k=1}^{\nu} \ln(PSVECN(k)/PSVECD(k))$

  where ν = `size(PSVECN)` = `size(PSVECD)`

- $range(:n) = \frac{1}{\sqrt{(2/edofn)+(2/edofd)}} (\max_{k=1}^{\nu} \ln(PSMATN(:n,k)/PSMATD(:n,k)) - \min_{k=1}^{\nu} \ln(PSMATN(:n,k)/PSMATD(:n,k))$

  where ν = `size(PSMATN,2)` = `size(PSMATD,2)` and n = `size(PSMATN,1)` = `size(PSMATD,1)` is the number of channels in the two multi-channel time series.

In both cases, ν is the number of frequencies considered. Arguments *EDOFN* and *EDOFD* give, respectively, the equivalent numbers of degrees of freedom, $edofn$ and $edofd$, of the first and second estimated spectra (e.g., the numerator and denominator of the ratio of the two estimated spectra). These numbers can be computed by the *estim_dof()* and *estim_dof2()* functions.

In order to derive approximate critical probability values, it is assumed that the elements of the vector $\ln(PSVECN(:)/PSVECD(:))$ (or of the vectors $\ln(PSMATN(i,:)/PSMATD(i,:))$ for i = 1 to n) are independent and follow approximately a normal distribution with mean $(1/edofn) - (1/edofd)$ and variance $(2/edofn) + (2/edofd)$. In these conditions, the distribution of the $range$ statistics may be approximated by the distribution function of the range of ν independent normal random variables (with mean and variance as specified above) as computed by the `rangen()` routine in the *Prob_Procedures* module [Potscher_Reschenhofer:1989].

The range log-ratio statistics $range$ are stored on output in the *RANGE* scalar or vector arguments.

This statistical test relies on the assumptions that the different spectral ordinates have the same sampling distribution and are independent of each other for each time series. This means, in particular, that the spectral ordinates corresponding to the zero and Nyquist frequencies must be excluded from the *PSVECN* and *PSVECD* vector arguments (or from the *PSMATN* and *PSMATD* matrix arguments) before calling **spctrm_ratio4()** and that the two estimated spectra have not been obtained by smoothing the periodogram in the frequency domain, but by averaging different periodograms computed on replicated time series.

Thus, this test could only be used to compare two periodograms or two spectral estimates computed as the average of, say, r periodograms for each time series.

It is also assumed that the (multichannel) time series with spectra *PSVECN* and *PSVECD* (or *PSMATN* and *PSMATD*) are independent realizations.

For more details and theory, see [Coates_Diggle:1986] [Potscher_Reschenhofer:1988] [Potscher_Reschenhofer:1989].

Synopsis:

```
call spctrm_ratio4( psvecn(:n)    , psvecd(:n)    , edofn , edofd , range_
↪stat     , prob     )
call spctrm_ratio4( psmatn(:p,:n) , psmatd(:p,:n) , edofn , edofd , range_
↪stat(:p) , prob(:p) )
```

spctrm_diff()

Purpose:

spctrm_diff() calculates approximate critical probability values (e.g., p-values) for testing the hypothesis of a common shape for two estimated (multi-channel) spectra (e.g., the vector arguments *PSVEC1* and *PSVEC2* or matrix arguments *PSMAT1*, *PSMAT2*). These approximate critical probability values are derived from the following Kolmogorov-Smirnov statistics (stored in the *KS_STAT* output scalar or vector arguments):

- $D = \sup_{m=1}^{\nu} |F1(m) - F2(m)|$

  where ν = `size(PSVEC1)` = `size(PSVEC2)`

- $D(j) = \sup_{m=1}^{\nu} |F1(j,m) - F2(j,m)|$ for j = 1 to n

  where ν = `size(PSMAT1,2)` = `size(PSMAT2,2)` and `n` = `size(PSMAT1,1)` = `size(PSMAT2,1)` is the number of channels in the two multi-channel time series.

In both cases, ν is the number of frequencies considered and F1() and F2() are the normalized cumulative periodograms computed from the estimated spectra *PSVEC1* and *PSVEC2* (or *PSMAT1* and *PSMAT2*).

The distribution of D under the null hypothesis of a common shape for the spectra of the two series is approximated by calculating D for some large number (e.g., the *NREP* argument) of random interchanges of periodogram ordinates at each frequency for the two estimated (multi-channel) spectra [Diggle_Fisher:1991].

This statistical randomization test relies on the assumptions that the different spectral ordinates have the same sampling distribution and are independent of each other [Priestley:1981]. This means, in particular, that the spectral ordinates corresponding to the zero and Nyquist frequencies must be excluded from the *PSVEC1* and *PSVEC2* vectors (or the *PSMAT1* and *PSMAT2* matrices) before calling **spctrm_diff()** and that the two estimated multichannel spectra have not been obtained by smoothing the periodograms in the frequency domain.

Thus, this randomization test could only be used to compare two periodograms or two spectral estimates computed as the average of, say, r periodograms for each time series.

For more details, see [Diggle_Fisher:1991].

Synopsis:

```
call spctrm_diff( psvec1(:n)    , psvec2(:n)    , ks_stat     , prob     ,␣
↪nrep=nrep , norm=norm , initseed=initseed )
call spctrm_diff( psmat1(:p,:n) , psmat2(:p,:n) , ks_stat(:p) , prob(:p) ,␣
↪nrep=nrep , norm=norm , initseed=initseed )
```

spctrm_diff2()

Purpose:

spctrm_diff2() calculates approximate critical probability values (e.g., p-values) for testing the hypothesis of a common underlying spectrum for the two estimated (multi-channel) spectra (e.g., the vector arguments *PSVEC1* and *PSVEC2* or matrix arguments *PSMAT1*, *PSMAT2*). These approximate critical probability values are derived from the following chi-squared log-ratio statistics (stored in the *CHI2_STAT* output scalar or vector arguments):

- $chi2 = \frac{1}{\nu} \sum_{k=1}^{\nu} \ln(PSVEC1(k)/PSVEC2(k))^2$

  where ν = `size(PSVEC1)` = `size(PSVEC2)`

- $chi2(:n) = \frac{1}{\nu} \sum_{k=1}^{\nu} \ln(PSMAT1(:n,k)/PSMAT2(:n,k))^2$

  where ν = `size(PSMAT1,2)` = `size(PSMAT2,2)` and n = `size(PSMAT1,1)` = `size(PSMAT2,1)` is the number of channels in the two multi-channel time series.

In both cases, ν is the number of frequencies considered.

The distribution of the chi-squared statistics $chi2$ under the null hypothesis of a common spectrum for the spectra of the two (multi-channel) time series is approximated by calculating the chi-squared statistic for some large number (e.g., the *NREP* argument) of random interchanges of periodogram ordinates at each frequency for the two estimated (multi-channel) spectra (e.g., the arguments *PSVEC1* and *PSVEC2* or *PSMAT1* and *PSMAT2*).

This statistical randomization test relies on the assumptions that the different spectral ordinates have the same sampling distribution and are independent of each other [Priestley:1981]. This means, in particular, that the spectral ordinates corresponding to the zero and Nyquist frequencies must be excluded from the PSVEC1 and PSVEC2 vectors (or *PSMAT1* and *PSMAT2* matrices) before calling **spctrm_diff2()** and that the two estimated (multi-channel) spectra have not been obtained by smoothing the periodograms in the frequency domain.

Thus, this randomization test could only be used to compare two periodograms or two spectral estimates computed as the average of, say, r periodograms for each time series.

Finally, none of the spectral estimates must be zero.

For more details, see [Diggle_Fisher:1991].

Synopsis:

```
call spctrm_diff2( psvec1(:n)    , psvec2(:n)    , chi2_stat     , prob     ,␣
↪nrep=nrep , initseed=initseed )
call spctrm_diff2( psmat1(:p,:n) , psmat2(:p,:n) , chi2_stat(:p) , prob(:p) ,␣
↪nrep=nrep , initseed=initseed )
```

power_spctrm()

Purpose:

power_spctrm() computes Fast Fourier Transform (FFT) estimates of the power spectrum of a real (multi-channel) time series (e.g., the vector argument *VEC* or matrix argument *MAT*). The real valued sequence time series must be of even length in all cases.

The Power Spectral Density (PSD) estimates are returned in units which are the square of the data (if *NORMPSD* = `false`) or in spectral density units (if *NORMPSD* = `true`).

After removing the mean or the trend from the (multi-channel) time series (e.g., the *TREND* argument), the selected data window (e.g., the *WIN* argument) is applied to the (multi-channel) time series and the PSD estimates are computed by the FFT of this transformed (multi-channel) time series. Optionally, theses PSD estimates may then be smoothed in the frequency domain by a Daniell filter (e.g., if the *NSMOOTH* argument is used).

For definitions, more details and algorithm, see [Bloomfield:1976], [Welch:1967] [Cooley_etal:1970] and [Diggle:1990].

Synopsis:

```
call power_spctrm( vec(:n)     , psvec(:(n/2)+1)     , freq=freq(:(n/2)+1) ,␣
↪fftvec=fftvec(:(n/2)+1)     , edof=edof(:(n/2)+1) , bandwidth=bandwidth(:(n/
↪2)+1) , conlwr=conlwr(:(n/2)+1) , conupr=conupr(:(n/2)+1) , initfft=initfft␣
↪, normpsd=normpsd , nsmooth=nsmooth , trend=trend , win=win , taperp=taperp␣
↪, probtest=probtest )
call power_spctrm( mat(:p,:n) , psmat(:p,:(n/2)+1) , freq=freq(:(n/2)+1) ,␣
↪fftmat=fftmat(:p,:(n/2)+1) , edof=edof(:(n/2)+1) , bandwidth=bandwidth(:(n/
↪2)+1) , conlwr=conlwr(:(n/2)+1) , conupr=conupr(:(n/2)+1) , initfft=initfft␣
↪, normpsd=normpsd , nsmooth=nsmooth , trend=trend , win=win , taperp=taperp␣
↪, probtest=probtest )
```

cross_spctrm()

Purpose:

cross_spctrm() computes Fast Fourier Transform (FFT) estimates of the power and cross-spectra of two real time series (e.g., the vector arguments *VEC* and *VEC2) or a real time series and a (multi-channel) time series (e.g., the vector argument *VEC* and matrix argument *MAT*). The real valued sequence time series must be of even length in all cases.

The Power Spectral Density (PSD) and Cross Spectral Density (CSD) estimates are returned in units which are the square of the data (if *NORMPSD* = `false`) or in spectral density units (if *NORMPSD* = `true`).

After removing the mean or the trend from the (multi-channel) time series (e.g., the *TREND* argument), the selected data window (e.g., the *WIN* argument) is applied to the (multi-channel) time series and the PSD and CSD estimates are computed by the FFT of this transformed (multi-channel) time series. Optionally, theses PSD estimates may then be smoothed in the frequency domain by a Daniell filter (e.g., if the *NSMOOTH* argument is used).

For definitions, more details and algorithm, see [Bloomfield:1976], [Welch:1967] [Cooley_etal:1970] and [Diggle:1990].

Synopsis:

```
call cross_spctrm( vec(:n) , vec2(:n)    , psvec(:(n/2)+1) , psvec2(:(n/
↪2)+1)    , phase(:(n/2)+1)     , coher(:(n/2)+1)     , freq=freq(:(n/2)+1) ,
↪ edof=edof(:(n/2)+1) , bandwidth=bandwidth(:(n/2)+1) , conlwr=conlwr(:(n/
↪2)+1) , conupr=conupr(:(n/2)+1) , testcoher=testcoher(:(n/2)+1) ,␣
↪ampli=ampli(:(n/2)+1)     , co_spect=co_spect(:(n/2)+1)     , quad_spect=quad_
↪spect(:(n/2)+1)     , prob_coher=prob_coher(:(n/2)+1)     , initfft=initfft ,␣
↪normpsd=normpsd , nsmooth=nsmooth , trend=trend , win=win , taperp=taperp ,␣
↪probtest=probtest )
call cross_spctrm( vec(:n) , mat(:p,:n) , psvec(:(n/2)+1) , psmat(:p,:(n/
↪2)+1) , phase(:p,:(n/2)+1) , coher(:p,:(n/2)+1) , freq=freq(:(n/2)+1) ,␣
↪edof=edof(:(n/2)+1) , bandwidth=bandwidth(:(n/2)+1) , conlwr=conlwr(:(n/
↪2)+1) , conupr=conupr(:(n/2)+1) , testcoher=testcoher(:(n/2)+1) ,␣
↪ampli=ampli(:p,:(n/2)+1) , co_spect=co_spect(:p,:(n/2)+1) , quad_spect=quad_
↪spect(:p,:(n/2)+1) , prob_coher=prob_coher(:p,:(n/2)+1) , initfft=initfft ,␣
```

```
↪normpsd=normpsd , nsmooth=nsmooth , trend=trend , win=win , taperp=taperp ,␣
↪probtest=probtest )
```

power_spctrm2()

Purpose:

power_spctrm2() computes Fast Fourier Transform (FFT) estimates of the power spectrum of a real (multi-channel) time series (e.g., the vector argument *VEC* or matrix argument *MAT*).

The Power Spectral Density (PSD) estimates are returned in units which are the square of the data (if *NORMPSD* = `false`) or in spectral density units (if *NORMPSD* = `true`).

After removing the mean or the trend from the (multi-channel) time series (e.g., the *TREND* argument), the time series are padded with zero on the right such that the length of the resulting augmented time series are evenly divisible by *L* (a positive even integer). The length, say `n`, of this resulting (multi-channel) time series is the first integer greater than or equal to `size(VEC)` (or `size(MAT,2)`) which is evenly divisible by *L*. If `size(VEC)` (or `size(MAT,2)`) is not evenly divisible by *L*, `n` is equal to `size(VEC)+L-mod(size(VEC),L)` (or `size(MAT,2)+L-mod(size(MAT,2),L)`.

Once the (multi-channel) time series has been segmented, the mean or the trend may also be removed from each (multi-channel) time segment (e.g., the *TREND2* argument), a data window (e.g., the *WIN* argument) is, eventually, applied to the (multi-channel) time segments. Optionally, zeros may also be added to each (multi-channel) time segment (e.g., the optional argument *L0*) if more finely spaced spectral estimates are desired [Welch:1967] [Cooley_etal:1970].

The PSD estimates are then derived by computing and averaging the FFTs of the transformed (multi-channel) time segments (e.g., modified periodograms). The stability of the PSD estimates depends on the averaging process. That is, the greater the number of segments (`n/L` if *OVERLAP* = `false` and `(2n/L)-1` if *OVERLAP* = `true`), the more stable the resulting PSD estimates [Welch:1967] [Cooley_etal:1970].

Optionally, theses PSD estimates may then be smoothed again in the frequency domain by a Daniell filter (e.g., if the *NSMOOTH* argument is used).

For definitions, more details and algorithm, see [Bloomfield:1976], [Welch:1967] [Cooley_etal:1970] and [Diggle:1990].

Synopsis:

```
call power_spctrm2( vec(:n)     , l , psvec(:((l+l0)/2)+1)  ␣
↪ , freq=freq(:((l+l0)/2)+1) , edof=edof(:((l+l0)/2)+1) ,␣
↪bandwidth=bandwidth(:((l+l0)/2)+1) , conlwr=conlwr(:((l+l0)/2)+1) ,␣
↪conupr=conupr(:((l+l0)/2)+1) , initfft=initfft , overlap=overlap ,␣
↪normpsd=normpsd , nsmooth=nsmooth , trend=trend , trend2=trend2 , win=win ,␣
↪taperp=taperp , l0=l0 , probtest=probtest )
call power_spctrm2( mat(:p,:n) , l , psmat(:p,:((l+l0)/2)+1)␣
↪, freq=freq(:((l+l0)/2)+1) , edof=edof(:((l+l0)/2)+1) ,␣
↪bandwidth=bandwidth(:((l+l0)/2)+1) , conlwr=conlwr(:((l+l0)/2)+1) ,␣
↪conupr=conupr(:((l+l0)/2)+1) , initfft=initfft , overlap=overlap ,␣
↪normpsd=normpsd , nsmooth=nsmooth , trend=trend , trend2=trend2 , win=win ,␣
↪taperp=taperp , l0=l0 , probtest=probtest )
```

cross_spctrm2()

Purpose:

cross_spctrm2() computes Fast Fourier Transform (FFT) estimates of the power and cross-spectra of two real time series (e.g., the vector arguments *VEC* and *VEC2) or a real time series and a (multi-channel) time series (e.g., the vector argument *VEC* and matrix argument *MAT*).

The Power Spectral Density (PSD) and Cross Spectral Density (CSD) estimates are returned in units which are the square of the data (if *NORMPSD* = `false`) or in spectral density units (if *NORMPSD* = `true`).

After removing the mean or the trend from the (multi-channel) time series (e.g., the *TREND* argument), the time series are padded with zero on the right such that the length of the resulting augmented time series are evenly divisible by *L* (a positive even integer). The length, say `n`, of this resulting (multi-channel) time series is the first integer greater than or equal to `size(VEC)` which is evenly divisible by *L*. If `size(VEC)` is not evenly divisible by *L*, `n` is equal to `size(VEC)+L-mod(size(VEC),L)`.

Once the (multi-channel) time series have been segmented, the mean or the trend may also be removed from each (multi-channel) time segment (e.g., the *TREND2* argument), a data window (e.g., the *WIN* argument) is, eventually, applied to the (multi-channel) time segments. Optionally, zeros may also be added to each (multi-channel) time segment (e.g., the optional argument *L0*) if more finely spaced spectral estimates are desired [Welch:1967] [Cooley_etal:1970].

The PSD and CSD estimates are then derived by computing and averaging the FFTs of the transformed (multi-channel) time segments (e.g., modified periodograms). The stability of the PSD and CSD estimates depends on the averaging process. That is, the greater the number of segments (`n/L` if *OVERLAP* = `false` and `(2n/L)-1` if *OVERLAP* = `true`), the more stable the resulting PSD estimates [Welch:1967] [Cooley_etal:1970].

Optionally, theses PSD and CSD estimates may then be smoothed again in the frequency domain by a Daniell filter (e.g., if the *NSMOOTH* argument is used).

For definitions, more details and algorithm, see [Bloomfield:1976], [Welch:1967] [Cooley_etal:1970] and [Diggle:1990].

Synopsis:

```
call cross_spctrm2( vec(:n) , vec2(:n)   , l , psvec(:((l+l0)/2)+1) ,␣
↪psvec2(:((l+l0)/2)+1)   , phase(:((l+l0)/2)+1)    , coher(:((l+l0)/
↪2)+1)    , freq=freq(:((l+l0)/2)+1) , edof=edof(:((l+l0)/2)+1) ,␣
↪bandwidth=bandwidth(:((l+l0)/2)+1) , conlwr=conlwr(:((l+l0)/2)+1) ,␣
↪conupr=conupr(:((l+l0)/2)+1) , testcoher=testcoher(:((l+l0)/2)+1) ,␣
↪ampli=ampli(:((l+l0)/2)+1)    , co_spect=co_spect(:((l+l0)/2)+1)    , quad_
↪spect=quad_spect(:((l+l0)/2)+1)    , prob_coher=prob_coher(:((l+l0)/2)+1)  ␣
↪ , initfft=initfft , overlap=overlap , normpsd=normpsd , nsmooth=nsmooth␣
↪, trend=trend , trend2=trend2 , win=win , taperp=taperp , l0=l0 ,␣
↪probtest=probtest )
call cross_spctrm2( vec(:n) , mat(:p,:n) , l , psvec(:((l+l0)/2)+1)␣
↪, psmat(:p,:((l+l0)/2)+1) , phase(:p,:((l+l0)/2)+1) , coher(:p,
↪:((l+l0)/2)+1) , freq=freq(:((l+l0)/2)+1) , edof=edof(:((l+l0)/2)+1)␣
↪, bandwidth=bandwidth(:((l+l0)/2)+1) , conlwr=conlwr(:((l+l0)/2)+1) ,
↪ conupr=conupr(:((l+l0)/2)+1) , testcoher=testcoher(:((l+l0)/2)+1) ,␣
↪ampli=ampli(:p,:((l+l0)/2)+1) , co_spect=co_spect(:p,:((l+l0)/2)+1) ,
↪ quad_spect=quad_spect(:p,:((l+l0)/2)+1) , prob_coher=prob_coher(:p,
↪:((l+l0)/2)+1) , initfft=initfft , overlap=overlap , normpsd=normpsd ,␣
↪nsmooth=nsmooth , trend=trend , trend2=trend2 , win=win , taperp=taperp ,
↪ l0=l0 , probtest=probtest )
```

power_spectrum()

Purpose:

power_spectrum() computes Fast Fourier Transform (FFT) estimates of the power spectrum of a real (multi-channel) time series (e.g., the vector argument *VEC* or matrix argument *MAT*). The real valued sequence time series must be of even length in all cases.

The Power Spectral Density (PSD) estimates are returned in units which are the square of the data (if *NORMPSD* = `false`) or in spectral density units (if *NORMPSD* = `true`).

After removing the mean or the trend from the (multi-channel) time series (e.g., the *TREND* argument), the selected data window (e.g., the *WIN* argument) is applied to the (multi-channel) time series and the PSD estimates are computed by the FFT of this transformed (multi-channel) time series. Optionally, theses PSD estimates may then be smoothed

in the frequency domain by application of modified Daniell filters (e.g., if the *SMOOTH_PARAM* vector argument is used) [Bloomfield:1976]. The use of modified Daniell filters instead of a simple Daniell filter for smoothing the periodogram is the main difference of **power_spectrum()** with *power_spctrm()* subroutine.

For definitions, more details and algorithm, see [Bloomfield:1976], [Welch:1967] [Cooley_etal:1970] and [Diggle:1990].

Synopsis:

```
call power_spectrum( vec(:n)     , psvec(:(n/2)+1)     , freq=freq(:(n/
↪2)+1) , fftvec=fftvec(:(n/2)+1)     , edof=edof , bandwidth=bandwidth ,
↪ conlwr=conlwr , conupr=conupr , initfft=initfft , normpsd=normpsd ,␣
↪smooth_param=smooth_param(:) , trend=trend , win=win , taperp=taperp ,␣
↪probtest=probtest )
call power_spectrum( mat(:p,:n) , psmat(:p,:(n/2)+1) , freq=freq(:(n/
↪2)+1) , fftmat=fftmat(:p,:(n/2)+1) , edof=edof , bandwidth=bandwidth ,
↪ conlwr=conlwr , conupr=conupr , initfft=initfft , normpsd=normpsd ,␣
↪smooth_param=smooth_param(:) , trend=trend , win=win , taperp=taperp ,␣
↪probtest=probtest )
```

Examples:

ex1_power_spectrum.F90

cross_spectrum()

Purpose:

cross_spectrum() computes Fast Fourier Transform (FFT) estimates of the power and cross-spectra of two real time series (e.g., the vector arguments *VEC* and *VEC2) or a real time series and a (multi-channel) time series (e.g., the vector argument *VEC* and matrix argument *MAT*). The real valued sequence time series must be of even length in all cases.

The Power Spectral Density (PSD) and Cross Spectral Density (CSD) estimates are returned in units which are the square of the data (if *NORMPSD* = `false`) or in spectral density units (if *NORMPSD* = `true`).

After removing the mean or the trend from the (multi-channel) time series (e.g., the *TREND* argument), the selected data window (e.g., the *WIN* argument) is applied to the (multi-channel) time series and the PSD and CSD estimates are computed by the FFT of this transformed (multi-channel) time series. Optionally, theses PSD estimates may then be smoothed in the frequency domain by application of modified Daniell filters (e.g., if the *SMOOTH_PARAM* vector argument is used) [Bloomfield:1976]. The use of modified Daniell filters instead of a simple Daniell filter for smoothing the periodogram is the main difference of **cross_spectrum()** with *cross_spctrm()* subroutine.

For definitions, more details and algorithm, see [Bloomfield:1976], [Welch:1967] [Cooley_etal:1970] and [Diggle:1990].

Synopsis:

```
call cross_spectrum( vec(:n) , vec2(:n)    , psvec(:(n/2)+1) , psvec2(:(n/
↪2)+1)    , phase(:(n/2)+1)     , coher(:(n/2)+1)     , freq=freq(:(n/2)+1)␣
↪, edof=edof , bandwidth=bandwidth , conlwr=conlwr , conupr=conupr ,␣
↪testcoher=testcoher , ampli=ampli(:(n/2)+1)     , co_spect=co_spect(:(n/
↪2)+1)     , quad_spect=quad_spect(:(n/2)+1)     , prob_coher=prob_coher(:(n/
↪2)+1)     , initfft=initfft , normpsd=normpsd , smooth_param=smooth_param(:)␣
↪, trend=trend , win=win , taperp=taperp , probtest=probtest )
call cross_spectrum( vec(:n) , mat(:p,:n) , psvec(:(n/2)+1) , psmat(:p,:(n/
↪2)+1) , phase(:p,:(n/2)+1) , coher(:p,:(n/2)+1) , freq=freq(:(n/2)+1)␣
↪, edof=edof , bandwidth=bandwidth , conlwr=conlwr , conupr=conupr ,␣
↪testcoher=testcoher , ampli=ampli(:p,:(n/2)+1) , co_spect=co_spect(:p,:(n/
↪2)+1) , quad_spect=quad_spect(:p,:(n/2)+1) , prob_coher=prob_coher(:p,:(n/
```

```
↪2)+1) , initfft=initfft , normpsd=normpsd , smooth_param=smooth_param(:) ,␣
↪trend=trend , win=win , taperp=taperp , probtest=probtest )
```

power_spectrum2()

Purpose:

power_spectrum2() computes Fast Fourier Transform (FFT) estimates of the power spectrum of a real (multi-channel) time series (e.g., the vector argument *VEC* or matrix argument *MAT*).

The Power Spectral Density (PSD) estimates are returned in units which are the square of the data (if *NORMPSD* = `false`) or in spectral density units (if *NORMPSD* = `true`).

After removing the mean or the trend from the (multi-channel) time series (e.g., the *TREND* argument), the time series are padded with zero on the right such that the length of the resulting augmented time series are evenly divisible by *L* (a positive even integer). The length, say `n`, of this resulting (multi-channel) time series is the first integer greater than or equal to `size(VEC)` (or `size(MAT,2)`) which is evenly divisible by *L*. If `size(VEC)` (or `size(MAT,2)`) is not evenly divisible by *L*, `n` is equal to `size(VEC)+L-mod(size(VEC),L)` (or `size(MAT,2)+L-mod(size(MAT,2),L)`.

Once the (multi-channel) time series has been segmented, the mean or the trend may also be removed from each (multi-channel) time segment (e.g., the *TREND2* argument), a data window (e.g., the *WIN* argument) is, eventually, applied to the (multi-channel) time segments. Optionally, zeros may also be added to each (multi-channel) time segment (e.g., the optional argument *L0*) if more finely spaced spectral estimates are desired [Welch:1967] [Cooley_etal:1970].

The PSD estimates are then derived by computing and averaging the FFTs of the transformed (multi-channel) time segments (e.g., modified periodograms). The stability of the PSD estimates depends on the averaging process. That is, the greater the number of segments (`n/L` if *OVERLAP* = `false` and `(2n/L)-1` if *OVERLAP* = `true`), the more stable the resulting PSD estimates [Welch:1967] [Cooley_etal:1970].

Optionally, theses PSD estimates may then be smoothed again in the frequency domain by modified Daniell filters (e.g., if the *SMOOTH_PARAM* argument is used) [Bloomfield:1976]. The use of modified Daniell filters instead of a simple Daniell filter for smoothing the PSD estimates is the main difference of **power_spectrum2()** with the *power_spctrm2()* subroutine.

For definitions, more details and algorithm, see [Bloomfield:1976], [Welch:1967] [Cooley_etal:1970] and [Diggle:1990].

Synopsis:

```
call power_spectrum2( vec(:n)     , l , psvec(:((l+l0)/2)+1)     ,␣
↪freq=freq(:((l+l0)/2)+1) , edof=edof , bandwidth=bandwidth , conlwr=conlwr␣
↪, conupr=conupr , initfft=initfft , overlap=overlap , normpsd=normpsd ,
↪ smooth_param=smooth_param(:) , trend=trend , trend2=trend2 , win=win ,␣
↪taperp=taperp , l0=l0 , probtest=probtest )
call power_spectrum2( mat(:p,:n) , l , psmat(:p,:((l+l0)/2)+1) ,␣
↪freq=freq(:((l+l0)/2)+1) , edof=edof , bandwidth=bandwidth , conlwr=conlwr␣
↪, conupr=conupr , initfft=initfft , overlap=overlap , normpsd=normpsd ,
↪ smooth_param=smooth_param(:) , trend=trend , trend2=trend2 , win=win ,␣
↪taperp=taperp , l0=l0 , probtest=probtest )
```

cross_spectrum2()

Purpose:

cross_spectrum2() computes Fast Fourier Transform (FFT) estimates of the power and cross-spectra of two real time series (e.g., the vector arguments *VEC* and *VEC2) or a real time series and a (multi-channel) time series (e.g., the vector argument *VEC* and matrix argument *MAT*).

The Power Spectral Density (PSD) and Cross Spectral Density (CSD) estimates are returned in units which are the square of the data (if *NORMPSD* = `false`) or in spectral density units (if *NORMPSD* = `true`).

After removing the mean or the trend from the (multi-channel) time series (e.g., the *TREND* argument), the time series are padded with zero on the right such that the length of the resulting augmented time series are evenly divisible by *L* (a positive even integer). The length, say `n`, of this resulting (multi-channel) time series is the first integer greater than or equal to `size(VEC)` which is evenly divisible by *L*. If `size(VEC)` is not evenly divisible by *L*, `n` is equal to `size(VEC)+L-mod(size(VEC),L)`.

Once the (multi-channel) time series have been segmented, the mean or the trend may also be removed from each (multi-channel) time segment (e.g., the *TREND2* argument), a data window (e.g., the *WIN* argument) is, eventually, applied to the (multi-channel) time segments. Optionally, zeros may also be added to each (multi-channel) time segment (e.g., the optional argument *L0*) if more finely spaced spectral estimates are desired [Welch:1967] [Cooley_etal:1970].

The PSD and CSD estimates are then derived by computing and averaging the FFTs of the transformed (multi-channel) time segments (e.g., modified periodograms). The stability of the PSD and CSD estimates depends on the averaging process. That is, the greater the number of segments (`n/L` if *OVERLAP* = `false` and `(2n/L)-1` if *OVERLAP* = `true`), the more stable the resulting PSD estimates [Welch:1967] [Cooley_etal:1970].

Optionally, theses PSD and CSD estimates may then be smoothed again in the frequency domain by modified Daniell filters [Bloomfield:1976]. The use of modified Daniell filters instead of a simple Daniell filter for smoothing the PSD and CSD estimates is the main difference of **cross_spectrum2()** with the *cross_spctrm2()* subroutine.

For definitions, more details and algorithm, see [Bloomfield:1976], [Welch:1967] [Cooley_etal:1970] and [Diggle:1990].

Synopsis:

```
call cross_spectrum2( vec(:n) , vec2(:n)    , l , psvec(:((l+l0)/2)+1) ,
↪ psvec2(:((l+l0)/2)+1)    , phase(:((l+l0)/2)+1)     , coher(:((l+l0)/
↪2)+1)     , freq=freq(:((l+l0)/2)+1) , edof=edof , bandwidth=bandwidth ,␣
↪conlwr=conlwr , conupr=conupr , testcoher=testcoher , ampli=ampli(:((l+l0)/
↪2)+1)     , co_spect=co_spect(:((l+l0)/2)+1)     , quad_spect=quad_
↪spect(:((l+l0)/2)+1)     , prob_coher=prob_coher(:((l+l0)/2)+1)     ,␣
↪initfft=initfft , overlap=overlap , normpsd=normpsd , smooth_param=smooth_
↪param(:) , trend=trend , trend2=trend2 , win=win , taperp=taperp , l0=l0 ,␣
↪probtest=probtest )
call cross_spectrum2( vec(:n) , mat(:p,:n) , l , psvec(:((l+l0)/2)+1) ,␣
↪psmat(:p,:((l+l0)/2)+1) , phase(:p,:((l+l0)/2)+1) , coher(:p,:((l+l0)/
↪2)+1) , freq=freq(:((l+l0)/2)+1) , edof=edof , bandwidth=bandwidth ,␣
↪conlwr=conlwr , conupr=conupr , testcoher=testcoher , ampli=ampli(:p,
↪:((l+l0)/2)+1) , co_spect=co_spect(:p,:((l+l0)/2)+1) , quad_spect=quad_
↪spect(:p,:((l+l0)/2)+1) , prob_coher=prob_coher(:p,:((l+l0)/2)+1) ,␣
↪initfft=initfft , overlap=overlap , normpsd=normpsd , smooth_param=smooth_
↪param(:) , trend=trend , trend2=trend2 , win=win , taperp=taperp , l0=l0 ,␣
↪probtest=probtest )
```

5.28 MODULE BLAS_interfaces

Module *BLAS_interfaces* contains/exports generic interfaces for selected routines available in the BLAS library for use inside of the STATPACK library when the cpp macro `_BLAS` is activated at compilation of the STATPACK library.

Use of these interface blocks exported by module *BLAS_interfaces* unsures that calls to BLAS routines are correct, when used inside the STATPACK library.

Since the BLAS library provides routines only for single and double precision real/complex data, the interface blocks defined in the module *BLAS_interfaces* will work obviously only if the real/complex kind type **stnd** defined in module *Select_Parameters* is equivalent to single or double precision real/complex data. In other words, you cannot activate BLAS support in STATPACK with the cpp macro `_BLAS` if the real/complex kind type **stnd** defined in module

Select_Parameters is equivalent to quadruple precision real/complex data because current versions of the BLAS library do not support quadruple precision real/complex data.

Generic interfaces are presently provided for the following BLAS routines:

- BLAS1 subroutines:

  axpy()
  Generic interface for SAXPY, DAXPY, CAXPY and ZAXPY subroutines (add vectors, y = a.x + y)

  copy()
  Generic interface for SCOPY, DCOPY, CCOPY and ZCOPY subroutines (copy vector, y = x)

  dot()
  Generic interface for SDOT, DDOT, CDOTC and ZDOTC subroutines (dot product, x^H y)

  dotu()
  Generic interface for CDOTU and ZDOTU subroutines (dot product, unconjugated x^T y)

  rot()
  Generic interface for SROT, DROT, CSROT and ZDROT subroutines (apply Givens plane rotation)

  swap()
  Generic interface for SSWAP, DSWAP, CSWAP and ZSWAP subroutines (swap vectors)

  scal()
  Generic interface for SSCAL, DSCAL, CSCAL, ZSCAL, CSSCAL and ZDSCAL subroutines (scale vector, y = a.y)

  nrm2()
  Generic interface for SNRM2, DNRM2, SCNRM2 and DZNRM2subroutines (vector 2-norm, $||x||_2$)

- BLAS2 subroutines:

  gemv()
  Generic interface for SGEMV, DGEMV, CGEMV and ZGEMV subroutines (matrix-vector multiply, y = a.Ax + b.y)

  symv()
  Generic interface for SSYMV and DSYMV subroutines (matrix-vector multiply, symmetric, y = a.Ax + b.y)

  spmv()
  Generic interface for SSPMV and DSPMV subroutines (matrix-vector multiply, symmetric packed, y = a.Ax + b.y)

  ger()
  Generic interface for SGER, DGER, CGERC and ZGERC subroutines (rank 1 update, conjugated, A = a.xy^H + A)

  geru()
  Generic interface for CGERU and ZGERU subroutines (rank 1 update, unconjugated, A = a.xy^T + A)

  trsv()
  Generic interface for STRSV, DTRSV, CTRSV and ZTRSV subroutines (triangular solve Tx = b)

her2()
Generic interface for CHER2 and ZHER2 subroutines (rank 2 update, conjugated, A = a.xy^H + conj(a).yx^H + A)

syr2()
Generic interface for SSYR2 and DSYR2 subroutines (rank 2 update, A = a.xy^T + a.yx^T + A)

hpr2()
Generic interface for CHPR2 and ZHPR2 subroutines (rank 2 update, hermitian packed, A = a.xy^H + conj(a).yx^H + A)

spr2()
Generic interface for SSPR2 and DSPR2 subroutines (rank 2 update, symmetric packed, A = a.xy^T + a.yx^T + A)

- BLAS3 subroutines:

gemm()
Generic interface for SGEMM, DGEMM, CGEMM and ZGEMM subroutines (matrix-matrix multiply, C = a.AB + b.C)

symm()
Generic interface for SSYMM, DSYMM, CSYMM and ZSYMM subroutines (matrix-matrix multiply, symmetric, C = a.AB + b.C)

hemm()
Generic interface for CHEMM and ZHEMM subroutines (matrix-matrix multiply, hermitian, C = a.AB + b.C)

trmm()
Generic interface for STRMM, DTRMM, CTRMM and ZTRMM subroutines (matrix-matrix multiply, triangular, C = a.AB + b.C)

trsm()
Generic interface for STRSM, DTRSM, CTRSM and ZTRSM subroutines (triangular solve multiple rhs, triangular, TX = a.B)

syrk()
Generic interface for SSYRK, DSYRK, CSYRK and ZSYRK subroutines (rank k update, symmetric, C = a.AA^T + b.C)

herk()
Generic interface for CHERK and ZHERK subroutines (rank k update, hermitian, C = a.AA^H + b.C)

syr2k()
Generic interface for SSYR2K, DSYR2K, CSYR2K and ZSYR2K subroutines (rank 2k update, symmetric, C = a.AB^T + a.BA^T + b.C)

her2k()
Generic interface for CHER2K and ZHER2K subroutines (rank 2k update, hermitian, C = a.AB^H + conj(a).BA^H + b.C)

Consult the official BLAS site at BLAS, the hyper-text documentation at BLAS documentation or the nice summary available at http://www.icl.utk.edu/~mgates3/docs/lapack.html for the definition/documentation of the BLAS routines.

Finally, note that you can add at your convenience interface blocks for other BLAS routines in module *BLAS_interfaces*, which is in the file `Module_BLAS_Interfaces.F90`. Here, is an example of the generic interface for the SDOT, DDOT, CDOTC and ZDOTC functions available in BLAS, which can be used as a model for creating a generic interface for other BLAS routines:

```
!
!   Interface for DOT functions in BLAS
!
interface dot
!
    REAL function sdot( N, SX, INCX, SY, INCY )
!     ..
!     .. Scalar Arguments ..
      INTEGER           INCX, INCY, N
!     ..
!     .. Array Arguments ..
      REAL              SX( * ), SY( * )
    end function
!
    DOUBLE PRECISION function ddot( N, DX, INCX, DY, INCY )
!     ..
!     .. Scalar Arguments ..
      INTEGER           INCX, INCY, N
!     ..
!     .. Array Arguments ..
      DOUBLE PRECISION DX( * ), DY( * )
    end function
!
    COMPLEX function cdotc( N, CX, INCX, CY, INCY )
!     ..
!     .. Scalar Arguments ..
      INTEGER           INCX, INCY, N
!     ..
!     .. Array Arguments ..
      COMPLEX           CX( * ), CY( * )
    end function
!
    COMPLEX*16 function zdotc( N, ZX, INCX, ZY, INCY )
!     ..
!     .. Scalar Arguments ..
      INTEGER           INCX, INCY, N
!     ..
!     .. Array Arguments ..
      COMPLEX*16        ZX( * ), ZY( * )
    end function
!
end interface
```

The following example programs illustrate the use of the *BLAS_interfaces* module in the framework of STATPACK:

Examples:

ex1_random_svd_with_blas.F90

ex1_random_eig_with_blas.F90

ex1_random_eig_pos_with_blas.F90

5.29 MODULE Lapack_interfaces

Module *Lapack_interfaces* contains/exports generic interfaces for selected routines/drivers available in the LAPACK library (LAPACK) for use within the framework of STATPACK. Note, however, that contrary to the BLAS routines,

which are used inside the STATPACK library if the cpp macro _BLAS is activated at compilation of STATPACK, LAPACK routines are not presently used inside STATPACK and the cpp macro _LAPACK is not defined in STATPACK and will have no effect at compilation.

However, use of the interface blocks exported by module *Lapack_interfaces* unsures that calls to LAPACK routines are correct, when used with STATPACK. Generic interfaces are presently provided for the following LAPACK routines and drivers:

- Tridiagonal reduction of a real symmetric or complex hermitian matrix:

  sytrd()
  Generic interface for SSYTRD, DSYTRD, CHETRD and ZHETRD subroutines (decomposition)

  Examples: ex1_lapack_sytrd.F90 ex2_lapack_sytrd.F90

  orgtr()
  Generic interface for SORGTR, CORGTR, CUNGTR and ZUNGTR subroutines (generation of orthogonal matrix)

  Examples: ex1_lapack_orgtr.F90

  ormtr()
  Generic interface for SORMTR, CORMTR, CUNMTR and ZUNMTR subroutines (multiplication by orthogonal matrix)

  Examples: ex1_lapack_ormtr.F90 ex2_lapack_ormtr.F90

- Eigenvalues and eigenvectors decomposition of a real symmetric or complex hermitian matrix:

  syev()
  Generic interface for SSYEV, DSYEV, CHEEV and ZHEEV subroutines (implicit QR/QL method)

  Examples: ex1_lapack_syev.F90 ex2_lapack_syev.F90

  syevd()
  Generic interface for SSYEVD, DSYEVD, CHEEVD and ZHEEVD subroutines (divide and conquer method)

  Examples: ex1_lapack_syevd.F90 ex2_lapack_syevd.F90

  syevr()
  Generic interface for SSYEVR, DSYEVR, CHEEVR and ZHEEVR subroutines (MRRR method)

  Examples: ex1_lapack_syevr.F90 ex2_lapack_syevr.F90 ex3_lapack_syevr.F90

  syevx()
  Generic interface for SSYEVX, DSYEVX, CHEEVX and ZHEEVX subroutines (bisection and inverse iteration)

  Examples: ex1_lapack_syevx.F90 ex2_lapack_syevx.F90 ex3_lapack_syevx.F90

- Eigenvalues and eigenvectors decomposition of a real symmetric or complex hermitian matrix in packed storage:

  spev()
  Generic interface for SSPEV, DSPEV, CHPEV and ZHPEV subroutines (implicit QR/QL method)

  Examples: ex1_lapack_spev.F90

spevd()
Generic interface for SSPEVD, DSPEVD, CHPEVD and ZHPEVD subroutines (divide and conquer method)

Examples: ex1_lapack_spevd.F90

spevx()
Generic interface for SSPEVX, DSPEVX, CHPEVX and ZHPEVX subroutines (bisection and inverse iteration)

Examples: ex1_lapack_spevx.F90 ex3_lapack_spevx.F90

- Eigenvalues and eigenvectors decomposition of a real symmetric tridiagonal matrix:

steqr()
Generic interface for SSTEQR, DSTEQR, CSTEQR and ZSTEQR subroutines (implicit QR/QL method)

stedc()
Generic interface for SSTEDC, DSTEDC, CSTEDC and ZSTEDC subroutines (divide and conquer method)

stemr()
Generic interface for SSTEMR, DSTEMR, CSTEMR and ZSTEMR subroutines (MRRR method)

Examples: ex1_lapack_stemr.F90 ex2_lapack_stemr.F90 ex3_lapack_stemr.F90

stevx()
Generic interface for SSTEVX and DSTEVX subroutines (partial or full spectrum by bisection and inverse iteration)

stev()
Generic interface for SSTEV and DSTEV subroutines (eigenvalues by the Pal-Walker-Kahan variant of the QL or QR algorithm or eigenvalues/eigenvectors by implicit QR/QL method)

stevd()
Generic interface for SSTEVD and DSTEVD subroutines (eigenvalues by the Pal-Walker-Kahan variant of the QL or QR algorithm or eigenvalues/eigenvectors by divide and conquer method)

stevr()
Generic interface for SSTEVR and DSTEVR subroutines (full spectrum by MRRR method and partial spectrum by bisection and inverse iteration)

- Eigenvalues and eigenvectors of a real generalized symmetric-definite or complex generalized hermitian-definite problem:

sygv()
Generic interface for SSYGV, DSYGV, CHEGV and ZHEGV subroutines (implicit QR/QL method)

sygvd()
Generic interface for SSYGVD, DSYGVD, CHEGVD and ZHEGVD subroutines (divide and conquer method)

sygvx()
Generic interface for SSYGVX, DSYGVX, CHEGVX and ZHEGVX subroutines (bisection and inverse iteration)

- Eigenvalues and eigenvectors of a real or complex general matrix:

geev()
Generic interface for SGEEV, DGEEV, CGEEV and ZGEEV subroutines (QR method)

geevx()
Generic interface for SGEEVX, DGEEVX, CGEEVX and ZGEEVX subroutines (QR method with balancing)

- Bidiagonal reduction of a real or complex general matrix:

gebrd()
Generic interface for SGEBRD, DGEBRD, CGEBRD and ZGEBRD subroutines (decomposition)

Examples: ex1_lapack_gebrd.F90 ex2_lapack_gebrd.F90

orgbr()
Generic interface for SORGBR, DORGBR, CUNGBR and ZUNGBR subroutines (generation of orthogonal matrices)

Examples: ex1_lapack_orgbr.F90

ormbr()
Generic interface for SORMBR, CORMBR, CUNMBR and ZUNMBR subroutines (multiplication by orthogonal matrices)

Examples: ex1_lapack_ormbr.F90 ex2_lapack_ormbr.F90

- Singular Value Decomposition (SVD) of a real or complex general matrix:

gesvd()
Generic interface for SGESVD, DGESVD, CGESVD and ZGESVD subroutines (implicit QR method)

Examples: ex1_lapack_gesvd.F90 ex2_lapack_gesvd.F90 ex3_lapack_gesvd.F90

gesdd()
Generic interface for SGESDD, DGESDD, CGESDD and ZGESDD subroutines (divide and conquer method)

Examples: ex1_lapack_gesdd.F90 ex2_lapack_gesdd.F90 ex3_lapack_gesdd.F90

gesvdx()
Generic interface for SGESVDX, DGESVDX, CGESVDX and ZGESVDX SGESVD subroutines (bisection/inverse iteration, including partial SVD)

Examples: ex1_lapack_gesvdx.F90 ex2_lapack_gesvdx.F90 ex3_lapack_gesvdx.F90

- Singular Value Decomposition (SVD) of a real bidiagonal matrix:

bdsqr()
Generic interface for SBDSQR, DBDSQR, CBDSQR and ZBDSQR subroutines (implicit QR method)

bdsdc()
Generic interface for SBDSDC and DBDSDC subroutines (divide and conquer method)

bdsvdx()
Generic interface for SBDSVDX and DBDSVDX subroutines (bisection/inverse iteration, including partial SVD)

- Solution of a real or complex system of linear equations with a general matrix and several right hand side vectors:

gesv()
Generic interface for SGESV, DGESV, CGESV and ZGESV subroutines (LU decomposition)

Examples: ex1_lapack_gesv.F90 ex2_lapack_gesv.F90

- Solution of a real or complex system of linear equations with a symmetric matrix and several right hand side vectors:

  sysv()
  Generic interface for SSYSV, DSYSV, CSYSV and ZSYSV subroutines (diagonal pivoting method)

  Examples: ex1_lapack_sysv.F90 ex2_lapack_sysv.F90

- Solution of a real or complex system of linear equations with a symmetric or hermitian positive definite matrix and several right hand side vectors:

  posv()
  Generic interface for SPOSV, DPOSV, CPOSV and ZPOSV subroutines (Cholesky decomposition)

  Examples: ex1_lapack_posv.F90 ex2_lapack_posv.F90

- Minimum-norm solution of a real or complex linear least square problem with several right hand side vectors:

  gelss()
  Generic interface for SGELSS, DGELSS, CGELSS and ZGELSS subroutines (SVD via implicit QR method)

  Examples: ex1_lapack_gelss.F90 ex2_lapack_gelss.F90

  gelsd()
  Generic interface for SGELSD, DGELSD, CGELSD and ZGELSD subroutines (SVD via divide and conquer method)

  Examples: ex1_lapack_gelsd.F90 ex2_lapack_gelsd.F90

  gelsy()
  Generic interface for SGELSY, DGELSY, CGELSY and ZGELSY subroutines (complete orthogonal factorization)

  Examples: ex1_lapack_gelsy.F90 ex2_lapack_gelsy.F90

- Solution of a real or complex, overdetermined or underdetermined, linear system with a coefficient matrix of full rank and several right hand side vectors:

  gels()
  Generic interface for SGELS, DGELS, CGELS and ZGELS subroutines (QR/QL method)

  Examples: ex1_lapack_gels.F90 ex2_lapack_gels.F90

- Computation of an unblocked QR factorization of a real or complex matrix:

  geqr2()
  Generic interface for SGEQR2, DGEQR2, CGEQR2 and ZGEQR2 subroutines

- Computation of a blocked QR factorization of a real or complex matrix:

  geqrf()
  Generic interface for SGEQRF, DGEQRF, CGEQRF and ZGEQRF subroutines

- Computation of a blocked QR factorization with column pivoting of a real or complex matrix using level 3 BLAS:

  geqp3()
  Generic interface for SGEQP3, DGEQP3, CGEQP3 and ZGEQP3 subroutines

- Computation of a blocked QR factorization of a "triangular-pentagonal" real or complex matrix, which is composed of a triangular block A and pentagonal block B:

tpqrt()
Generic interface for STPQRT, DTPQRT, CTPQRT, ZTPQRT subroutines

- Application of an orthogonal matrix Q obtained from a "triangular-pentagonal" block reflector to a general real or complex matrix:

tpmqrt()
Generic interface for STPMQRT, DTPMQRT, CTPMQRT, ZTPMQRT subroutines

Consult the official LAPACK site at LAPACK, the hyper-text documentation at LAPACK documentation or the nice summary available at http://www.icl.utk.edu/~mgates3/docs/lapack.html for the definition/documentation of the LAPACK routines.

See the FORTRAN programs ex1_lapack_gesdd.F90 and ex2_lapack_gesdd.F90 for working examples of using the STATPACK Fortran 95 generic interface *gesdd()* (defined in the *Lapack_interfaces* module) for subroutines x**GESDD**() (where x can be **S**, **D**, **C** or **Z**) available in the LAPACK library for performing a Singular Value Decomposition (SVD) of a real/complex matrix of kind **stnd** by the divide and conquer method, inside the framework of STATPACK.

Since the LAPACK library provides routines only for single and double precision real/complex data, the interface blocks defined in the module *Lapack_interfaces* will work obviously only if the real/complex kind type **stnd** defined in module *Select_Parameters* is equivalent to single or double precision real/complex data.

Finally, note that you can add at your convenience interface blocks for other LAPACK routines in module *Lapack_interfaces*, which is in the file `Module_Lapack_Interfaces.F90`. Here, is an example of the generic interface *syevd()* for the SSYEVD, DSYEVD, CHEEVD and ZHEEVD subroutines available in LAPACK, which can be used as a model for creating a generic interface for other LAPACK subroutines:

```
!
!   Generic interface for SSYEVD, DSYEVD, CHEEVD and ZHEEVD subroutines in LAPACK
!
interface syevd
!
    subroutine ssyevd( JOBZ, UPLO, N, A, LDA, W, WORK, LWORK, IWORK, LIWORK, INFO )
!     ..
!     .. Scalar Arguments ..
       CHARACTER          JOBZ, UPLO
       INTEGER            INFO, LDA, LIWORK, LWORK, N
!     ..
!     .. Array Arguments ..
       INTEGER            IWORK( * )
       REAL               A( LDA, * ), W( * ), WORK( * )
    end subroutine
!
    subroutine dsyevd( JOBZ, UPLO, N, A, LDA, W, WORK, LWORK, IWORK, LIWORK, INFO )
!     ..
!     .. Scalar Arguments ..
       CHARACTER          JOBZ, UPLO
       INTEGER            INFO, LDA, LIWORK, LWORK, N
!     ..
!     .. Array Arguments ..
       INTEGER            IWORK( * )
       DOUBLE PRECISION A( LDA, * ), W( * ), WORK( * )
    end subroutine
!
    subroutine cheevd( JOBZ, UPLO, N, A, LDA, W, WORK, LWORK, RWORK, LRWORK, IWORK,␣
↪LIWORK, INFO )
!     ..
!     .. Scalar Arguments ..
```

(continues on next page)

(continued from previous page)

```
       CHARACTER          JOBZ, UPLO
       INTEGER            INFO, LDA, LIWORK, LRWORK, LWORK, N
!      ..
!      .. Array Arguments ..
       INTEGER            IWORK( * )
       REAL               W( * ), RWORK( * )
       COMPLEX            A( LDA, * ), WORK( * )
    end subroutine
!
    subroutine zheevd( JOBZ, UPLO, N, A, LDA, W, WORK, LWORK, RWORK, LRWORK, IWORK,␣
↪LIWORK, INFO )
!      ..
!      .. Scalar Arguments ..
       CHARACTER          JOBZ, UPLO
       INTEGER            INFO, LDA, LIWORK, LRWORK, LWORK, N
!      ..
!      .. Array Arguments ..
       INTEGER            IWORK( * )
       DOUBLE PRECISION W( * ), RWORK( * )
       COMPLEX*16         A( LDA, * ), WORK( * )
    end subroutine
!
end interface
```

5.30 MODULE Statpack

Module *Statpack* is an interface module, which exports all the constants, subroutines and functions publicly available from other modules available in the STATPACK library.

Using the *Statpack* module in your Fortran program is the simplest and standard way of accessing the routines and constants available in the STATPACK library.

Note that the *Statpack* module exports also interface blocks for several routines/drivers available in the BLAS and LAPACK libraries as defined in the modules *BLAS_interfaces* and *Lapack_interfaces*.

Since BLAS and LAPACK libraries provide routines only for single and double precision real/complex data, the interface blocks defined in the modules *BLAS_interfaces* and *Lapack_interfaces* and exported by the *Statpack* module will work obviously only if the real/complex kind type **stnd** defined in module *Select_Parameters* is equivalent to single or double precision real/complex data.

The *Statpack* module contains just `use` statements for the different STATPACK modules and a `public` statement for exporting all the public constants and routines from these modules:

```
!
! USED MODULES
! ============
!
use Select_Parameters, only : i1b, i2b, i4b, i8b, lgl, stnd, extd,     &
                              n1_def, n2_def, n3_def,                  &
                              defunit, urandom_file, blksz_util,       &
                              blksz_lin, blksz_fft, blksz_qr,          &
                              blksz_eig, blksz2_eig, blksz2_svd,       &
                              max_francis_steps_svd,                   &
                              max_francis_steps_eig,                   &
                              omp_limit, omp_limit2, omp_chunk,        &
```

(continues on next page)

(continued from previous page)

```
                                max_num_threads_symtrid_cmp,            &
                                max_num_threads_symtrid_cmp2,           &
                                max_num_threads_bd_cmp,                 &
                                max_num_threads_bd_cmp2,                &
                                npar_arth, npar2_arth,                  &
                                npar_geop, npar2_geop,                  &
                                npar_cumsum, npar_cumprod,              &
                                npar_poly, npar_polyterm
!
use Derived_Types, only     : sprs2_stnd, sprs2_stndc, sprs2_extd,      &
                              sprs2_extdc
!
use Logical_Constants
use Reals_Constants
use Num_Constants
use Char_Constants
use Utilities
use Utilities_With_Pnter
use Random
!
use String_Procedures
use Print_Procedures
use Time_Procedures
use Sort_Procedures
!
use FFT_Procedures
use Giv_Procedures
use Hous_Procedures
use Lin_Procedures
use Eig_Procedures
use QR_Procedures
use SVD_Procedures
use LLSQ_Procedures
use Prob_Procedures
use Stat_Procedures
use Mul_Stat_Procedures
use Time_Series_Procedures
!
use BLAS_interfaces
use Lapack_interfaces
!
! PUBLIC ENTITIES
! ===============
!
public
```

CHAPTER

SIX

STATPACK MODULES MANUALS

6.1 Module_BLAS_Interfaces

Copyright 2024 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING GENERIC INTERFACES FOR SELECTED SUBROUTINES AND FUNCTIONS IN THE BLAS LIBRARY.

THIS INTERFACE MODULE ENSURES THAT CALLS TO BLAS ROUTINES ARE CORRECT, WHEN USED WITH STATPACK.

GENERIC INTERFACES ARE PRESENTLY PROVIDED FOR THE FOLLOWING BLAS ROUTINES:

> Xaxpy, Xcopy, Xdot, Xdotu, Xrot, Xswap, Xscal, Xnrm2, Xgemv, Xsymv, Xspmv, Xger, Xgeru, Xtrsv, Xher2, Xsyr2, Xhpr2, Xspr2, Xgemm, Xsymm, Xhemm, Xtrmm, Xtrsm, Xsyrk, Xherk, Xsyr2k, Xher2k

WHERE X CAN BE s, d, c AND z. THE GENERIC INTERFACES HAVE THE FORM:

> axpy, copy, dot, dotu, rot, swap, scal, nrm2, gemv, symv, spmv, ger, geru, trsv, her2, syr2, hpr2, spr2, gemm, symm, hemm, trmm, trsm, syrk, herk, syr2k, her2k

LATEST REVISION : 11/03/2024

6.2 Module_Char_Constants

Copyright 2022 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING CHARACTER CONSTANTS, STRINGS AND ERROR MESSAGES FOR ROUTINES AVAILABLE IN STATPACK.

LATEST REVISION : 17/08/2022

6.3 Module_Derived_Types

Copyright 2018 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING DERIVED DATA TYPES FOR SPARSE REAL AND COMPLEX MATRICES OF KIND stnd AND extd.

THE AVAILABLE DERIVED DATA TYPES ARE DEFINED AS FOLLOW:

```
type sprs2_stnd
    integer(i4b) :: n, len
    real(stnd), dimension(:), pointer :: val
    integer(i4b), dimension(:), pointer :: irow
    integer(i4b), dimension(:), pointer :: jcol
end type sprs2_stnd
type sprs2_extd
```

```
        integer(i4b) :: n, len
        real(extd), dimension(:), pointer :: val
        integer(i4b), dimension(:), pointer :: irow
        integer(i4b), dimension(:), pointer :: jcol
    end type sprs2_extd
    type sprs2_stndc
        integer(i4b) :: n, len
        complex(stnd), dimension(:), pointer :: val
        integer(i4b), dimension(:), pointer :: irow
        integer(i4b), dimension(:), pointer :: jcol
    end type sprs2_stndc
    type sprs2_extdc
        integer(i4b) :: n, len
        complex(extd), dimension(:), pointer :: val
        integer(i4b), dimension(:), pointer :: irow
        integer(i4b), dimension(:), pointer :: jcol
    end type sprs2_extdc
```

LATEST REVISION : 06/06/2018

6.4 Module_Eig_Procedures

Copyright 2025 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING PROCEDURES FOR COMPUTING (SELECTED) EIGENVALUES AND/OR (SELECTED) EIGENVECTORS OF A SYMMETRIC (TRIDIAGONAL) MATRIX.

SUBROUTINES FOR COMPUTING A PARTIAL EIGENVALUE DECOMPOSITION OF SYMMETRIC MATRICES BASED ON RANDOMIZED ALGORITHMS ARE ALSO PROVIDED.

LATEST REVISION : 17/09/2025

6.4.1 subroutine symtrid_cmp (mat, d, e, store_q, upper)

Purpose

SYMTRID_CMP reduces a real n-by-n symmetric matrix MAT to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry:

- If UPPER = true : The leading n-by-n upper triangular part of MAT contains the upper triangular part of the symmetric matrix MAT, and the strictly lower triangular part of MAT is not referenced.
- If UPPER = false : The leading n-by-n lower triangular part of MAT contains the lower triangular part of the symmetric matrix MAT, and the strictly upper triangular part of MAT is not referenced.

On exit:

- If UPPER = true and STORE_Q = true : The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = false and STORE_Q = true : The leading n-by-n lower triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = true and STORE_Q = false : The leading n-by-n upper triangular part of MAT is destroyed.
- If UPPER = false and STORE_Q = false : The leading n-by-n lower triangular part of MAT is destroyed.

D (OUTPUT) real(stnd), dimension(:) The diagonal elements of the tridiagonal matrix T:

- D(i) = T(i,i).

The size of D must verify: size(D) = size(MAT, 1) = size(MAT, 2) = n .

E (OUTPUT) real(stnd), dimension(:) The off-diagonal elements of the tridiagonal matrix T:

- E(i) = T(i,i+1) = T(i+1,i)
- E(n) is arbitrary.

The size of E must verify: size(E) = size(MAT, 1) = size(MAT, 2) = n .

STORE_Q (INPUT) logical(lgl) On exit:

- If UPPER = true and STORE_Q = true : The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = false and STORE_Q = true : The leading n-by-n lower triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = true and STORE_Q = false : The leading n-by-n upper triangular part of MAT is destroyed.
- If UPPER = false and STORE_Q = false : The leading n-by-n lower triangular part of MAT is destroyed.

UPPER (INPUT) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER= true : Upper triangular is stored ;
- UPPER= false: Lower triangular is stored .

Further Details

If UPPER = true and STORE_Q = true, the matrix Q is represented as a product of elementary reflectors

Q = H(n-1) * . . . * H(2) * H(1).

Each H(i) has the form

H(i) = I + tau * v * v'

where tau is a real scalar, and v is a real vector with v(i+1:n) = 0 ; v(1:i) is stored on exit in MAT(1:i,i+1), and tau in MAT(i+1,i+1).

If UPPER = false and STORE_Q = true, the matrix Q is represented as a product of elementary reflectors

Q = H(1) * H(2) * . . . * H(n-1).

Each H(i) has the form

H(i) = I + tau * v * v'

where tau is a real scalar, and v is a real vector with v(1:i) = 0 ; v(i+1:n) is stored on exit in MAT(i+1:n,i), and tau in MAT(i,i).

The contents of MAT on exit are illustrated by the following examples with n = 5:

if UPPER = true and STORE_Q = true :

(xx v1 v2 v3 v4)

(yy t1 v2 v3 v4)

(yy yy t2 v3 v4)

(yy yy yy t3 v4)

(yy yy yy yy t4)

if UPPER = false and STORE_Q = true :

(t1 yy yy yy yy)

(v1 t2 yy yy yy)

(v1 v2 t3 yy yy)

(v1 v2 v3 t4 yy)

(v1 v2 v3 v4 xx)

where vi and ti denote an element of the vector v and the scalar tau defining H(i), respectively. xx = machhuge and is used by the subroutine ORTHO_GEN_SYMTRID in order to verify that SYMTRID_CMP has been called before ORTHO_GEN_SYMTRID. Elements yy are not modified by the subroutine.

This subroutine is adapted from the routine DSYTD2 in LAPACK. Note that this subroutine is not blocked and not parallelized.

For more details on the reduction algorithm used in SYMTRID_CMP, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.4.2 subroutine symtrid_cmp (mat, d, e, store_q)

Purpose

SYMTRID_CMP reduces a real n-by-n symmetric matrix MAT to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry:

- The leading n-by-n upper triangular part of MAT contains the upper triangular part of the symmetric matrix MAT, and the strictly lower triangular part of MAT is not referenced.

On exit:

- If STORE_Q = true : The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If STORE_Q = false : The leading n-by-n upper triangular part of MAT is destroyed.

D (OUTPUT) real(stnd), dimension(:) The diagonal elements of the tridiagonal matrix T:

- D(i) = T(i,i).

The size of D must verify: size(D) = size(MAT, 1) = size(MAT, 2) = n.

E (OUTPUT) real(stnd), dimension(:) The off-diagonal elements of the tridiagonal matrix T:

- E(i) = T(i,i+1) = T(i+1,i)
- E(n) is arbitrary.

The size of E must verify: size(E) = size(MAT, 1) = size(MAT, 2) = n.

STORE_Q (INPUT) logical(lgl) On exit:

- If STORE_Q = true : The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors and the lower triangular part of MAT is destroyed. See Further Details.
- If STORE_Q = false : The symmetric matrix MAT is destroyed.

Further Details

If STORE_Q = true, the matrix Q is represented as a product of elementary reflectors

Q = H(n-1) * . . . * H(2) * H(1).

Each H(i) has the form

H(i) = I + tau * v * v'

where tau is a real scalar, and v is a real vector with v(i+1:n) = 0 ; v(1:i) is stored on exit in MAT(1:i,i+1), and tau in MAT(i+1,i+1).

The contents of MAT on exit are illustrated by the following example with n = 5:

(xx v1 v2 v3 v4)

(yy t1 v2 v3 v4)

(yy yy t2 v3 v4)

(yy yy yy t3 v4)

(yy yy yy yy t4)

where vi and ti denote an element of the vector v and the scalar tau defining H(i), respectively. xx = machhuge and is used by the subroutine ORTHO_GEN_SYMTRID in order to verify that SYMTRID_CMP has been called before ORTHO_GEN_SYMTRID. Elements yy are not modified by the subroutine.

This subroutine is adapted from the routine DSYTD2 in LAPACK. An efficient blocked algorithm is used to reduced the n-by-n symmetric matrix MAT to tridiagonal form T. Furthermore, the computations are parallelized if OPENMP is used.

In other words, SYMTRID_CMP is much more efficient then SYMTRID_CMP with argument UPPER, which is not blocked and not parallelized.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

6.4.3 `subroutine symtrid_cmp ( matp, d, e, store_q, upper )`

Purpose

SYMTRID_CMP reduces a real n-by-n symmetric matrix MAT stored in packed form to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

Arguments

MATP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the upper or lower triangle of the symmetric matrix MAT, packed column-wise in a linear array. The j-th column of MAT is stored in the array MATP as follows:

- if UPPER = true, MATP(i + (j-1) * j/2) = MAT(i,j) for 1<=i<=j;
- if UPPER = false, MATP(i + (j-1) * (2*n-j)/2) = MAT(i,j) for j<=i<=n.

On exit:

- If UPPER = true and STORE_Q = true : The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = false and STORE_Q = true : The leading n-by-n lower triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = true and STORE_Q = false : The leading n-by-n upper triangular part of MAT is destroyed.

- If UPPER = false and STORE_Q = false : The leading n-by-n lower triangular part of MAT is destroyed.

The size of MATP must verify: size(MATP) = (n * (n+1)/2)

D (OUTPUT) real(stnd), dimension(:) The diagonal elements of the tridiagonal matrix T:

- D(i) = T(i,i).

The size of D must verify: size(D) = n .

E (OUTPUT) real(stnd), dimension(:) The off-diagonal elements of the tridiagonal matrix T:

- E(i) = T(i,i+1) = T(i+1,i)
- E(n) is arbitrary.

The size of E must verify: size(E) = n .

STORE_Q (INPUT) logical(lgl) On exit:

- If UPPER = true and STORE_Q = true : The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = false and STORE_Q = true : The leading n-by-n lower triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = true and STORE_Q = false : The leading n-by-n upper triangular part of MAT is destroyed.
- If UPPER = false and STORE_Q = false : The leading n-by-n lower triangular part of MAT is destroyed.

UPPER (INPUT) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangle of MAT is stored;
- UPPER = false: Lower triangle of MAT is stored.

Further Details

If UPPER = true and STORE_Q = true, the matrix Q is represented as a product of elementary reflectors

Q = H(n-1) * . . . * H(2) * H(1).

Each H(i) has the form

H(i) = I + tau * v * v'

where tau is a real scalar, and v is a real vector with v(i+1:n) = 0 ; v(1:i) and tau are stored on exit in MATP, overwriting MAT(1:i,i+1) and MAT(i+1,i+1), respectively.

If UPPER = false and STORE_Q = true, the matrix Q is represented as a product of elementary reflectors

Q = H(1) * H(2) * . . . * H(n-1).

Each H(i) has the form

H(i) = I + tau * v * v'

where tau is a real scalar, and v is a real vector with v(1:i) = 0 ; v(i+1:n) and tau are stored on exit in MATP, overwriting MAT(i+1:n,i) and MAT(i,i), respectively.

The contents of MATP on exit are illustrated by the following examples with n = 5:

if UPPER = true and STORE_Q = true, MAT is equal to :

(xx v1 v2 v3 v4)

(yy t1 v2 v3 v4)

(yy yy t2 v3 v4)

(yy yy yy t3 v4)

(yy yy yy yy t4)

if UPPER = false and STORE_Q = true, MAT is equal to :

(t1 yy yy yy yy)

(v1 t2 yy yy yy)

(v1 v2 t3 yy yy)

(v1 v2 v3 t4 yy)

(v1 v2 v3 v4 xx)

where vi and ti denote an element of the vector v and the scalar tau defining H(i), respectively. Elements yy are not used and not stored in MATP. xx = machhuge and is used by other subroutines in order to verify that SYMTRID_CMP has been called.

This subroutine is adapted from the routine DSPTRD in LAPACK. Note that this subroutine is not blocked and not parallelized.

For more details on the reduction algorithm used in SYMTRID_CMP, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.4.4 subroutine symtrid_cmp (matp, d, e, store_q)

Purpose

SYMTRID_CMP reduces a real n-by-n symmetric matrix MAT stored in packed form to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

Arguments

MATP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the upper triangle of the symmetric matrix MAT, packed column-wise in a linear array. The j-th column of MAT is stored in the array MATP as follows:

MATP(i + (j-1) * j/2) = MAT(i,j) for 1<=i<=j;

On exit:

- If STORE_Q = true : The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If STORE_Q = false : The leading n-by-n upper triangular part of MAT is destroyed.

The size of MATP must verify: size(MATP) = (n * (n+1)/2)

D (OUTPUT) real(stnd), dimension(:) The diagonal elements of the tridiagonal matrix T:

- D(i) = T(i,i).

The size of D must verify: size(D) = n .

E (OUTPUT) real(stnd), dimension(:) The off-diagonal elements of the tridiagonal matrix T:

- E(i) = T(i,i+1) = T(i+1,i)
- E(n) is arbitrary.

The size of E must verify: size(E) = n .

STORE_Q (INPUT) logical(lgl) On exit:

- If STORE_Q = true : The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If STORE_Q = false : The leading n-by-n upper triangular part of MAT is destroyed.

Further Details

The matrix Q is represented as a product of elementary reflectors

Q = H(n-1) * . . . * H(2) * H(1).

Each H(i) has the form

H(i) = I + tau * v * v'

where tau is a real scalar, and v is a real vector with v(i+1:n) = 0 ; v(1:i) and tau are stored on exit in MATP, overwriting MAT(1:i,i+1) and MAT(i+1,i+1), respectively, if STORE_Q = true.

The contents of MATP (if STORE_Q = true) on exit are illustrated by the following example with n = 5 (giving the contents of MAT):

(xx v1 v2 v3 v4)

(yy t1 v2 v3 v4)

(yy yy t2 v3 v4)

(yy yy yy t3 v4)

(yy yy yy yy t4)

where vi and ti denote an element of the vector v and the scalar tau defining H(i), respectively. Elements yy are not used and not stored in MATP. xx = machhuge and is used by other subroutines in order to verify that SYMTRID_CMP has been called.

This subroutine is adapted from the routine DSPTRD in LAPACK. An efficient blocked algorithm is used to reduced the n-by-n symmetric matrix MAT to tridiagonal form T. Furthermore, the computations are parallelized if OPENMP is used.

In other words, SYMTRID_CMP is much more efficient then SYMTRID_CMP with argument UPPER which is not blocked and not parallelized.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

6.4.5 subroutine symtrid_cmp2 (mat, d, e, store_q)

Purpose

SYMTRID_CMP2 reduces a real n-by-n symmetric matrix product

MAT' * MAT

, where MAT is a m-by-n matrix with m >= n, to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT' * MAT * Q = T

where Q is an n-by-n orthogonal matrix.

SYMTRID_CMP2 computes T and Q, using the one-sided Ralha tridiagonal reduction algorithm without explicitly forming the matrix product MAT' * MAT .

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the general m-by-n matrix to be reduced.

On exit:

- If STORE_Q = true : The leading n-by-n lower triangular part of MAT is overwritten by the matrix Q as a product of elementary Householder reflectors. The other part of MAT is also destroyed.
- If STORE_Q = false : The m-by-n matrix MAT is destroyed.

The shape of MAT must verify: size(MAT, 1) >= size(MAT, 2) = n .

D (OUTPUT) real(stnd), dimension(:) The diagonal elements of the tridiagonal matrix T:

- D(i) = T(i,i).

The size of D must verify: size(D) = size(MAT, 1) = size(MAT, 2) = n .

E (OUTPUT) real(stnd), dimension(:) The off-diagonal elements of the tridiagonal matrix T:

- E(i) = T(i,i+1) = T(i+1,i)
- E(n) is arbitrary.

The size of E must verify: size(E) = size(MAT, 1) = size(MAT, 2) = n .

STORE_Q (INPUT) logical(lgl) On exit:

- If STORE_Q = true : The leading n-by-n lower triangular part of MAT is overwritten by the matrix Q as a product of elementary Householder reflectors.
- If STORE_Q = false : The m-by-n matrix MAT is destroyed.

  See Further details.

Further Details

This subroutine is an implementation of the Ralha one-sided method to reduce implicitly a matrix product MAT' * MAT to tridiagonal form T. Q is computed as a product of n-1 elementary reflectors (e.g., Householder transformations):

Q = H(1) * H(2) * . . . * H(n-1)

Each H(i) has the form:

H(i) = I + tau * v * v'

where tau is a real scalar, and v is a real vector. IF STORE_Q is set to true, the n-1 H(i) elementary reflectors are stored in the leading lower triangle of the array MAT. For the H(i) reflector, tau is stored in MAT(i,i) and v is stored in MATi+1:n,i). Note that if n is equal to 1, no elementary reflectors are needed.

This is the blocked version of the algorithm. See the references (1), (2) and (3) for further details. Furthermore the algorithm is parallelized if OPENMP is used.

The contents of MAT on exit are illustrated by the following example with n = 5 and m = 6:

if STORE_Q = true :

(t1 yy yy yy yy)

(v1 t2 yy yy yy)

(v1 v2 t3 yy yy)

(v1 v2 v3 t4 yy)

(v1 v2 v3 v4 xx)

(yy yy yy yy yy)

where vi and ti denote an element of the vector v and the scalar tau defining H(i), respectively. xx = machhuge and is used by the subroutine ORTHO_GEN_SYMTRID in order to verify that SYMTRID_CMP2 has been called before ORTHO_GEN_SYMTRID. Elements yy are destroyed by the subroutine.

Subroutines ORTHO_GEN_SYMTRID or APPLY_Q_SYMTRID, with logical argument UPPER set to false, can be used to generate the orthogonal matrix Q or to apply it to another matrix. See descriptions of these subroutines for further details.

For further details, see:

(1) **Hegland, M., Kahn, M., and Osborn, M., 1999:** A parallel algorithm for the reduction to tridiagonal form for eigendecomposition. SIAM Journal on Scientific Computing, 21:3, pp. 987-1005.

(2) **Ralha, R.M.S., 2003:** One-sided reduction to bidiagonal form. Linear Algebra Appl., No 358, pp. 219-238.

(3) **Barlow, J.L., Bosner, N., and Drmac, Z., 2005:** A new stable bidiagonal reduction algorithm. Linear Algebra Appl., No 397, pp. 35-84.

(4) **Bosner, N., and Barlow, J.L., 2007:** Block and Parallel versions of one-sided bidiagonalization. SIAM J. Matrix Anal. Appl., Volume 29, No 3, pp. 927-953.

6.4.6 subroutine ortho_gen_symtrid (mat, upper)

Purpose

ORTHO_GEN_SYMTRID generates a real orthogonal matrix Q, which is defined as the product of n-1 elementary reflectors of order n, as returned by SYMTRID_CMP with STORE_Q = true:

- if UPPER = true, Q = H(n-1) * ... * H(2) * H(1),
- if UPPER = false, Q = H(1) * H(2) * ... * H(n-1).

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the vectors and the scalars, which define the elementary reflectors, as returned by SYMTRID_CMP with STORE_Q = true.

On exit, the n-by-n orthogonal matrix Q.

UPPER (INPUT) logical(lgl) If:

- UPPER= true : Upper triangle of MAT contains elementary reflectors from SYMTRID_CMP;
- UPPER = false: Lower triangle of MAT contains elementary reflectors from SYMTRID_CMP.

Further Details

This subroutine is adapted from the routine SORGTR in LAPACK. A blocked algorithm is used for agregating the Householder transformations (e.g. the elementary reflectors) stored in the upper or lower triangle of MAT and generating the orthogonal matrix Q. Furthermore, the computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(3) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.4.7 `subroutine ortho_gen_symtrid ( mat )`

Purpose

ORTHO_GEN_SYMTRID generates a real orthogonal matrix Q, which is defined as the product of n-1 elementary reflectors of order n, as returned by SYMTRID_CMP with STORE_Q = true:

Q = H(n-1) * … * H(2) * H(1)

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the vectors and the scalars, which define the elementary reflectors, as returned by SYMTRID_CMP with STORE_Q = true.

On exit, the n-by-n orthogonal matrix Q.

Further Details

This subroutine is adapted from the routine SORGTR in LAPACK. A blocked algorithm is used for agregating the Householder transformations (e.g. the elementary reflectors) stored in the upper triangle of MAT and generating the orthogonal matrix Q. Furthermore, the computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(3) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.4.8 `subroutine apply_q_symtrid ( mat, c, left, trans, upper )`

Purpose

APPLY_Q_SYMTRID overwrites the general real m-by-n matrix C with:

- Q * C if LEFT = true and TRANS = false, or
- Q'* C if LEFT = true and TRANS = true, or
- C * Q if LEFT = false and TRANS = false, or
- C * Q' if LEFT = false and TRANS = true,

where Q is a real orthogonal matrix of order nq and is defined as the product of nq-1 elementary reflectors:

- Q = H(nq-1) * ... * H(2) * H(1), if UPPER = true
- Q = H(1) * H(2) * ... * H(nq-1), if UPPER = false

as returned by SYMTRID_CMP with STORE_Q = true.

Q is of order m (=nq) and is the product of m-1 reflectors if LEFT = true ; Q is of order n (=nq) and is the product of n-1 reflectors if LEFT = false.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the vectors and the scalars, which define the elementary reflectors, as returned by SYMTRID_CMP with STORE_Q = true. MAT is not modified by the routine.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = nq.

C (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m by n matrix C.

On exit, C is overwritten by Q * C or Q' * C or C * Q' or C * Q.

The shape of C must verify:

- if LEFT = true, size(C, 1) = nq ;
- if LEFT = false, size(C, 2) = nq .

LEFT (INPUT) logical(lgl) If:

- LEFT = true : apply Q or Q' from the left ;
- LEFT = false : apply Q or Q' from the right .

TRANS (INPUT) logical(lgl) If:

- TRANS = false : apply Q (no transpose) ;
- TRANS = true : apply Q' (transpose) .

UPPER (INPUT) logical(lgl) If:

- UPPER = true : The upper triangle of MAT contains elementary reflectors generated by SYMTRID_CMP;
- UPPER = false: The lower triangle of MAT contains elementary reflectors generated by SYMTRID_CMP.

Further Details

This subroutine is adapted from the routine SORMTR in LAPACK. A blocked algorithm is used to apply the Householder transformations (e.g. the elementary reflectors) stored in the upper or lower triangle of MAT (see the references (2) and (3) below).

Furthermore, the subroutine is parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(3) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.4.9 `subroutine apply_q_symtrid ( mat, c, left, trans )`

Purpose

APPLY_Q_SYMTRID overwrites the general real m-by-n matrix C with

- Q * C if LEFT = true and TRANS = false, or
- Q'* C if LEFT = true and TRANS = true, or
- C * Q if LEFT = false and TRANS = false, or
- C * Q' if LEFT = false and TRANS = true,

where Q is a real orthogonal matrix of order nq and is defined as the product of nq-1 elementary reflectors

Q = H(nq-1) * ... * H(2) * H(1)

as returned by SYMTRID_CMP (with UPPER = true or without this argument) and with STORE_Q = true.

Q is of order m (=nq) and is the product of m-1 reflectors if LEFT = true ; Q is of order n (=nq) and is the product of n-1 reflectors if LEFT = false.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the vectors and the scalars, which define the elementary reflectors, as returned by SYMTRID_CMP (with UPPER = true or SYMTRID_CMP without argument UPPER) and with STORE_Q = true. MAT is not modified by the routine.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = nq.

C (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m by n matrix C.

On exit, C is overwritten by Q * C or Q' * C or C * Q' or C * Q.

The shape of C must verify:

- if LEFT = true, size(C, 1) = nq ;
- if LEFT = false, size(C, 2) = nq .

LEFT (INPUT) logical(lgl) If:

- LEFT = true : apply Q or Q' from the left ;
- LEFT = false : apply Q or Q' from the right .

TRANS (INPUT) logical(lgl) If:

- TRANS = false : apply Q (no transpose) ;
- TRANS = true : apply Q' (transpose) .

Further Details

This subroutine is adapted from the routine SORMTR in LAPACK. A blocked algorithm is used to apply the Householder transformations (e.g. the elementary reflectors) stored in the upper triangle of MAT (see the references (2) and (3) below).

Furthermore, the subroutine is parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(3) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.4.10 `subroutine apply_q_symtrid ( matp, c, left, trans, upper )`

Purpose

APPLY_Q_SYMTRID overwrites the general real m-by-n matrix C with

- Q * C if LEFT = true and TRANS = false, or
- Q'* C if LEFT = true and TRANS = true, or
- C * Q if LEFT = false and TRANS = false, or
- C * Q' if LEFT = false and TRANS = true,

where Q is a real orthogonal matrix of order nq and is defined as the product of nq-1 elementary reflectors

- Q = H(nq-1) * ... * H(2) * H(1), if UPPER = true
- Q = H(1) * H(2) * ... * H(nq-1), if UPPER = false

as returned by SYMTRID_CMP with STORE_Q = true.

Q is of order m (=nq) and is the product of m-1 reflectors if LEFT = true ; Q is of order n (=nq) and is the product of n-1 reflectors if LEFT = false.

Arguments

MATP (INPUT) real(stnd), dimension(:) On entry, the vectors and the scalars, which define the elementary reflectors, as returned by SYMTRID_CMP in argument MATP. MATP is not modified by the routine.

The size of MATP must verify size(MATP) = (nq * (nq+1)/2)

C (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m by n matrix C.

On exit, C is overwritten by Q * C or Q' * C or C * Q' or C * Q.

The shape of C must verify:

- if LEFT = true, size(C, 1) = nq ;
- if LEFT = false, size(C, 2) = nq .

LEFT (INPUT) logical(lgl) If:

- LEFT = true : apply Q or Q' from the left ;
- LEFT = false : apply Q or Q' from the right .

TRANS (INPUT) logical(lgl) If:

- TRANS = false : apply Q (no transpose) ;
- TRANS = true : apply Q' (transpose) .

UPPER (INPUT) logical(lgl) Specifies whether the upper or lower triangular part of the original symmetric matrix MAT was stored in packed form in MATP before the reduction by SYMTRID_CMP. If:

- UPPER = true : Upper triangle of MAT was stored;
- UPPER = false: Lower triangle of MAT was stored.

Further Details

This subroutine is adapted from the routine DORMTR in LAPACK and uses a blocked algorithm to apply the Householder transformations (e.g. the elementary reflectors) stored in packed form in the vector MATP.

Furthermore, the subroutine is parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(3) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.4.11 subroutine apply_q_symtrid (matp, c, left, trans)

Purpose

APPLY_Q_SYMTRID overwrites the general real m-by-n matrix C with

- Q * C if LEFT = true and TRANS = false, or
- Q'* C if LEFT = true and TRANS = true, or
- C * Q if LEFT = false and TRANS = false, or
- C * Q' if LEFT = false and TRANS = true,

where Q is a real orthogonal matrix of order nq and is defined as the product of nq-1 elementary reflectors

- Q = H(nq-1) * ... * H(2) * H(1)

as returned by SYMTRID_CMP (with UPPER = true or without this argument) and with STORE_Q = true.

Q is of order m (=nq) and is the product of m-1 reflectors if LEFT = true ; Q is of order n (=nq) and is the product of n-1 reflectors if LEFT = false.

Arguments

MATP (INPUT) real(stnd), dimension(:) On entry, the vectors and the scalars, which define the elementary reflectors, as returned by SYMTRID_CMP in argument MATP. MATP is not modified by the routine.

The size of MATP must verify size(MATP) = (nq * (nq+1)/2)

C (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m by n matrix C.

On exit, C is overwritten by Q * C or Q' * C or C * Q' or C * Q.

The shape of C must verify:

- if LEFT = true, size(C, 1) = nq ;
- if LEFT = false, size(C, 2) = nq .

LEFT (INPUT) logical(lgl) If:

- LEFT = true : apply Q or Q' from the left ;
- LEFT = false : apply Q or Q' from the right .

TRANS (INPUT) logical(lgl) If:

- TRANS = false : apply Q (no transpose) ;
- TRANS = true : apply Q' (transpose) .

Further Details

This subroutine is adapted from the routine DORMTR in LAPACK and uses a blocked algorithm to apply the Householder transformations (e.g. the elementary reflectors) stored in packed form in the vector MATP.

Furthermore, the subroutine is parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(3) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.4.12 `subroutine eig_sort ( sort, d, u )`

Purpose

Given the eigenvalues D and eigenvectors U as output from EIG_CMP, EIG_CMP2 EIG_CMP3 or SYMTRID_QRI, SYMTRID_QRI2 and SYMTRID_QRI3, this routine sorts the eigenvalues into ascending or descending order, and, rearranges the columns of U correspondingly.

Arguments

SORT (INPUT) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The eigenvectors are reordered accordingly.

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the eigenvalues.

On exit, the eigenvalues in ascending or decreasing order.

U (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the columns of U are the eigenvectors.

On exit, U contains the reordered eigenvectors.

The shape of U must verify: size(U, 2) = size(D) = m .

Further Details

The method is straight insertion.

6.4.13 `subroutine eig_abs_sort ( sort, d, u )`

Purpose

Given the eigenvalues D and eigenvectors U as output from EIG_CMP, EIG_CMP2 EIG_CMP3 or SYMTRID_QRI, SYMTRID_QRI2 and SYMTRID_QRI3, this routine sorts the eigenvalues into ascending or descending order of absolute magnitude, and, rearranges the columns of U correspondingly.

Arguments

SORT (INPUT) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd' of absolute magnitude. The eigenvectors are reordered accordingly.

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the eigenvalues.

On exit, the eigenvalues in ascending or decreasing order of absolute magnitude.

U (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the columns of U are the eigenvectors.

On exit, U contains the reordered eigenvectors.

The shape of U must verify: size(U, 2) = size(D) = m .

Further Details

The method is straight insertion.

6.4.14 subroutine eigval_sort (sort, d)

Purpose

Given the eigenvalues D as output from EIGVAL_CMP, EIGVAL_CMP2, EIGVAL_CMP3 or SYMTRID_QRI, SYMTRID_QRI2 and SYMTRID_QRI3, this routine sorts the eigenvalues into ascending or descending order.

Arguments

SORT (INPUT) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the eigenvalues.

On exit, the eigenvalues in ascending or decreasing order.

Further Details

The method is quick sort.

6.4.15 subroutine eigval_abs_sort (sort, d)

Purpose

Given the eigenvalues D as output from EIGVAL_CMP, EIGVAL_CMP2, EIGVAL_CMP3 or SYMTRID_QRI, SYMTRID_QRI2 and SYMTRID_QRI3, this routine sorts the eigenvalues into ascending or descending order of absolute magnitude.

Arguments

SORT (INPUT) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd' of absolute magnitude.

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the eigenvalues.

On exit, the eigenvalues in ascending or decreasing order of absolute magnitude.

Further Details

The method is straight insertion.

6.4.16 subroutine symtrid_qri (d, e, failure, mat, init_mat, sort, maxiter)

Purpose

SYMTRID_QRI computes all eigenvalues and eigenvectors of a symmetric n-by-n tridiagonal matrix using the implicit QR method.

The eigenvalues and eigenvectors of a full symmetric matrix can also be found if SYMTRID_CMP and ORTHO_GEN_SYMTRID have been used to reduce this matrix to tridiagonal form before calling SYMTRID_QRI.

Arguments

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the diagonal elements of the tridiagonal matrix.

On exit, the eigenvalues.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the tridiagonal matrix. E(n) is arbitrary and is used as workspace.

On exit, E has been destroyed.

The size of E must verify: size(E) = size(D) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of the tridiagonal matrix.

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, if:

- INIT_MAT is absent or if INIT_MAT = false, then MAT contains the orthogonal matrix used in the reduction to tridiagonal form as returned by ORTHO_GEN_SYMTRID.
- INIT_MAT is present and INIT_MAT = true, MAT is set to the identity matrix of order n.

On exit, if FAILURE = false:

- MAT contains the orthonormal eigenvectors of the original symmetric matrix if INIT_MAT is absent or if INIT_MAT = false,
- MAT contains the orthonormal eigenvectors of the symmetric tridiagonal matrix if INIT_MAT is present and INIT_MAT = true.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = size(D) = n .

INIT_MAT (INPUT, OPTIONAL) logical(lgl) If:

- INIT_MAT = false: The subroutine computes eigenvalues and eigenvectors of the original symmetric matrix. On entry, MAT must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.
- INIT_MAT = true: The subroutine computes eigenvalues and eigenvectors of the tridiagonal matrix. MAT is initialized to the identity matrix of order n.

The default is false.

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The eigenvectors are reordered accordingly.

By default, the eigenvalues are not sorted.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of the tridiagonal matrix. The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * n. Convergence usually occurs in about 2 * n QR sweeps.

The default is 30.

Further Details

The eigenvalues and eigenvectors are computed by the implicit tridiagonal QR algorithm described in the reference (1) with modifications suggested in the reference (2).

This subroutine is adapted from the routine DSTEQR in LAPACK. Note that this subroutine is parallelized with OPENMP using the method described in the reference (3).

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Greenbaum, A., and J. Dongarra, J., 1989:** Experiments with QR/QL Methods for the Symmetric Tridiagonal Eigenproblem. LAPACK Working Note No 17, November 1989.

(3) **Demmel, J., Heath, M.T., and Van Der Vorst, H., 1993:** Parallel numerical linear algebra. Acta Numerica 2, 111-197.

6.4.17 `subroutine symtrid_qri ( d, e, failure, sort, maxiter )`

Purpose

SYMTRID_QRI computes all eigenvalues of a symmetric n-by-n tridiagonal matrix using the Pal-Walker-Kahan variant of the QR algorithm.

The eigenvalues of a full symmetric matrix can also be found if SYMTRID_CMP has been used to reduce this matrix to tridiagonal form before calling SYMTRID_QRI.

Arguments

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the diagonal elements of the tridiagonal matrix.

On exit, the eigenvalues.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the tridiagonal matrix. E(n) is arbitrary and is used as workspace.

On exit, E has been destroyed.

The size of E must verify: size(E) = size(D) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : this indicates successful exit.

- FAILURE = true : this indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of the tridiagonal matrix.

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

By default, the eigenvalues are not sorted.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of the tridiagonal matrix. The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * n. Convergence usually occurs in about 2 * n QR sweeps.

The default is 30.

Further Details

The eigenvalues are computed by the Pal-Walker-Kahan variant of the implicit tridiagonal QR algorithm described in the reference (1) with modifications suggested in the reference (2).

This subroutine is adapted from the routine DSTERF in LAPACK. This subroutine is not parallelized.

For further details, see:

(1) **Parlett, B.N., 1998:** The Symmetric Eigenvalue Problem. Revised edition, SIAM, Philadelphia.

(2) **Greenbaum, A., and J. Dongarra, J., 1989:** Experiments with QR/QL Methods for the Symmetric Tridiagonal Eigenproblem. LAPACK Working Note No 17, November 1989.

6.4.18 `subroutine symtrid_qri2 ( d, e, failure, mat, init_mat, sort, maxiter, max_francis_steps )`

Purpose

SYMTRID_QRI2 computes all eigenvalues and eigenvectors of a symmetric n-by-n tridiagonal matrix with a perfect shift strategy for the eigenvectors.

The eigenvalues and eigenvectors of a full symmetric matrix can also be found if SYMTRID_CMP and ORTHO_GEN_SYMTRID have been used to reduce this matrix to tridiagonal form before calling SYMTRID_QRI2.

Arguments

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the diagonal elements of the tridiagonal matrix.

On exit, the eigenvalues.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the tridiagonal matrix. E(n) is arbitrary and is used as workspace.

On exit, E has been destroyed.

The size of E must verify: size(E) = size(D) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.

- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of the tridiagonal matrix.

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, if:

- INIT_MAT is absent or if INIT_MAT = false, then MAT contains the orthogonal matrix used in the reduction to tridiagonal form as returned by ORTHO_GEN_SYMTRID.
- INIT_MAT is present and INIT_MAT = true, MAT is set to the identity matrix of order n.

On exit, if FAILURE = false:

- MAT contains the orthonormal eigenvectors of the original symmetric matrix if INIT_MAT is absent or if INIT_MAT = false,
- MAT contains the orthonormal eigenvectors of the symmetric tridiagonal matrix if INIT_MAT is present and INIT_MAT = true.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = size(D) = n .

INIT_MAT (INPUT, OPTIONAL) logical(lgl) If:

- INIT_MAT = false: The subroutine computes eigenvalues and eigenvectors of the original symmetric matrix. On entry, MAT must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.
- INIT_MAT = true: The subroutine computes eigenvalues and eigenvectors of the tridiagonal matrix. MAT is initialized to the identity matrix of order n.

The default is false.

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The eigenvectors are reordered accordingly.

By default, the eigenvalues are not sorted.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of the tridiagonal matrix. The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * n. Convergence usually occurs in about 2 * n QR sweeps.

The default is 30.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the eigenvectors in the QR algorithm. MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of n and the integer parameter MAX_FRANCIS_STEPS_EIG specified in the module Select_Parameters.

Further Details

The eigenvalues are computed by the Pal-Walker-Kahan variant of the implicit tridiagonal QR algorithm described in the reference (1).

The eigenvectors are computed with a perfect shift strategy (see the references (1) and (2)) with modifications suggested in the references (3) and (4) for deflating a given eigenvalue from the tridiagonal matrix, and modifications described in references (5) and (6) for applying a set of Givens rotations.

Furthermore, the computation of the eigenvectors is parallelized if OPENMP is used.

With all these changes, SYMTRID_QRI2 is much more efficient than SYMTRID_QRI for computing the full set of eigenvectors of a real n-by-n symmetric tridiagonal matrix for large matrices.

For further details, see:

(1) **Parlett, B.N., 1998:** The Symmetric Eigenvalue Problem. Revised edition, SIAM, Philadelphia.

(2) **Greenbaum, A., and J. Dongarra, J., 1989:** Experiments with QR/QL Methods for the Symmetric Tridiagonal Eigenproblem. LAPACK Working Note No 17, November 1989.

(3) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, 1013-1034.

(4) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

(5) **Lang, B., 1998:** Using level 3 BLAS in rotation-based algorithms. Siam J. Sci. Comput., Vol. 19, 626-634.

(6) **Van Zee, F.G., Van de Geijn, R., and Quintana-Orti, G., 2011:** Restructuring the QR Algorithm for High-Performance Application of Givens Rotations. FLAME Working Note 60. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-11-36.

6.4.19 `subroutine symtrid_qri2 ( d, e, failure, sort, maxiter )`

Purpose

SYMTRID_QRI2 computes all eigenvalues of a symmetric n-by-n tridiagonal matrix using the Pal-Walker-Kahan variant of the QR algorithm.

The eigenvalues of a full symmetric matrix can also be found if SYMTRID_CMP has been used to reduce this matrix to tridiagonal form before calling SYMTRID_QRI2.

Arguments

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the diagonal elements of the tridiagonal matrix.

On exit, the eigenvalues.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the tridiagonal matrix. E(n) is arbitrary and is used as workspace.

On exit, E has been destroyed

The size of E must verify: size(E) = size(D) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of the tridiagonal matrix.

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

By default, the eigenvalues are not sorted.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of the tridiagonal matrix. The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * n. Convergence usually occurs in about 2 * n QR sweeps.

The default is 30.

Further Details

The eigenvalues are computed by the Pal-Walker-Kahan variant of the implicit tridiagonal QR algorithm described in the reference (1).

This subroutine is adapted from the routine DSTERF in LAPACK. This subroutine is not parallelized.

For further details, see:

(1) **Parlett, B.N., 1998:** The Symmetric Eigenvalue Problem. revised edition, SIAM, Philadelphia.

6.4.20 subroutine symtrid_qri3 (d, e, failure, mat, init_mat, sort, maxiter, max_francis_steps)

Purpose

SYMTRID_QRI3 computes all eigenvalues and eigenvectors of a symmetric n-by-n tridiagonal matrix using the implicit QR method.

The eigenvalues and eigenvectors of a full symmetric matrix can also be found if SYMTRID_CMP and ORTHO_GEN_SYMTRID have been used to reduce this matrix to tridiagonal form before calling SYMTRID_QRI3.

Arguments

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the diagonal elements of the tridiagonal matrix.

On exit, the eigenvalues.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the tridiagonal matrix. E(n) is arbitrary and is used as workspace.

On exit, E has been destroyed.

The size of E must verify: size(E) = size(D) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of the tridiagonal matrix.

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, if:

- INIT_MAT is absent or if INIT_MAT = false, then MAT contains the orthogonal matrix used in the reduction to tridiagonal form as returned by ORTHO_GEN_SYMTRID.
- INIT_MAT is present and INIT_MAT = true, MAT is set to the identity matrix of order n.

On exit, if FAILURE = false:

- MAT contains the orthonormal eigenvectors of the original symmetric matrix if INIT_MAT is absent or if INIT_MAT = false,
- MAT contains the orthonormal eigenvectors of the symmetric tridiagonal matrix if INIT_MAT is present and INIT_MAT = true.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = size(D) = n .

INIT_MAT (INPUT, OPTIONAL) logical(lgl) If:

- INIT_MAT = false: The subroutine computes eigenvalues and eigenvectors of the original symmetric matrix. On entry, MAT must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.
- INIT_MAT = true: The subroutine computes eigenvalues and eigenvectors of the tridiagonal matrix. MAT is initialized to the identity matrix of order n.

The default is false.

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The eigenvectors are reordered accordingly.

By default, the eigenvalues are not sorted.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of the tridiagonal matrix. The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * n. Convergence usually occurs in about 2 * n QR sweeps.

The default is 30.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the eigenvectors in the QR algorithm. MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of n and the integer parameter MAX_FRANCIS_STEPS_EIG specified in the module Select_Parameters.

Further Details

The eigenvalues and eigenvectors are computed by the implicit tridiagonal QR algorithm described in the reference (1).

This subroutine is adapted from the routine DSTEQR in LAPACK with modifications suggested in the references (2) and (3) for the application of a set of Givens rotations.

Furthermore, the computations of the eigenvectors are parallelized if OPENMP is used.

SYMTRID_QRI3 is much more efficient than SYMTRID_QRI and only slightly less efficient than SYMTRID_QRI2 (but more robust) for computing the full set of eigenvectors of a real n-by-n symmetric tridiagonal matrix.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lang, B., 1998:** Using level 3 BLAS in rotation-based algorithms. Siam J. Sci. Comput., Vol. 19, 626-634.

(3) **Van Zee, F.G., Van de Geijn, R., and Quintana-Orti, G., 2011:** Restructuring the QR Algorithm for High-Performance Application of Givens Rotations. FLAME Working Note 60. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-11-36.

6.4.21 `subroutine symtrid_qri3 ( d, e, failure, sort, maxiter )`

Purpose

SYMTRID_QRI3 computes all eigenvalues of a symmetric n-by-n tridiagonal matrix using the implicit QR method. The eigenvalues of a full symmetric matrix can also be found if SYMTRID_CMP has been used to reduce this matrix to tridiagonal form before calling SYMTRID_QRI3.

Arguments

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the diagonal elements of the tridiagonal matrix.

On exit, the eigenvalues.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the tridiagonal matrix. E(n) is arbitrary and is used as workspace.

On exit, E has been destroyed.

The size of E must verify: size(E) = size(D) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of the tridiagonal matrix.

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

By default, the eigenvalues are not sorted.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of the tridiagonal matrix. The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * n. Convergence usually occurs in about 2 * n QR sweeps.

The default is 30.

Further Details

The eigenvalues are computed by the implicit tridiagonal QR algorithm described in the reference (1).

This subroutine is adapted from the routine DSTEQR in LAPACK. This subroutine is not parallelized.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.4.22 subroutine lae2 (a, b, c, rt1, rt2)

Purpose

LAE2 computes the eigenvalues of a 2-by-2 symmetric matrix

[A B]

[B C]

On return, RT1 is the eigenvalue of larger absolute value, and RT2 is the eigenvalue of smaller absolute value.

Arguments

A (INPUT) real(stnd) The (1,1) element of the 2-by-2 matrix.

B (INPUT) real(stnd) The (1,2) and (2,1) elements of the 2-by-2 matrix.

C (INPUT) real(stnd) The (2,2) element of the 2-by-2 matrix.

RT1 (OUTPUT) real(stnd) The eigenvalue of larger absolute value.

RT2 (OUTPUT) real(stnd) The eigenvalue of smaller absolute value.

Further Details

RT1 is accurate to a few ulps barring over/underflow.

RT2 may be inaccurate if there is massive cancellation in the determinant A * C - B * B; higher precision or correctly rounded or correctly truncated arithmetic would be needed to compute RT2 accurately in all cases.

Overflow is possible only if RT1 is within a factor of 5 of overflow. Underflow is harmless if the input data is 0 or exceeds underflow_threshold / macheps.

This subroutine is translated from the routine DLAE2 in LAPACK.

6.4.23 subroutine laev2 (a, b, c, rt1, rt2, cs1, sn1)

Purpose

LAEV2 computes the eigendecomposition of a 2-by-2 symmetric matrix

[A B]

[B C]

On return, RT1 is the eigenvalue of larger absolute value, RT2 is the eigenvalue of smaller absolute value, and (CS1,SN1) is the unit right eigenvector for RT1, giving the decomposition

[+CS1 +SN1] [A B] [+CS1 -SN1] = [RT1 000]

[-SN1 +CS1] [B C] [+SN1 +CS1] _ [000 RT2].

Arguments

A (INPUT) real(stnd) The (1,1) element of the 2-by-2 matrix.

B (INPUT) real(stnd) The (1,2) and (2,1) elements of the 2-by-2 matrix.

C (INPUT) real(stnd) The (2,2) element of the 2-by-2 matrix.

RT1 (OUTPUT) real(stnd) The eigenvalue of larger absolute value.

RT2 (OUTPUT) real(stnd) The eigenvalue of smaller absolute value.

CS1 (OUTPUT) real(stnd)

SN1 (OUTPUT) real(stnd) The vector (CS1, SN1) is a unit right eigenvector for RT1.

Further Details

RT1 is accurate to a few ulps barring over/underflow.

RT2 may be inaccurate if there is massive cancellation in the determinant A * C - B * B; higher precision or correctly rounded or correctly truncated arithmetic would be needed to compute RT2 accurately in all cases.

CS1 and SN1 are accurate to a few ulps barring over/underflow.

Overflow is possible only if RT1 is within a factor of 5 of overflow. Underflow is harmless if the input data is 0 or exceeds underflow_threshold / macheps.

This subroutine is translated from the routine DLAE2 in LAPACK.

6.4.24 subroutine symtrid_ratqri (d, e, m, failure, small, tol)

Purpose

SYMTRID_RATQRI computes the m largest or smallest eigenvalues of a symmetric n-by-n tridiagonal matrix using a rational QR method.

The m largest or smallest eigenvalues of a full symmetric matrix can also be found if SYMTRID_CMP has been used to reduce this matrix to tridiagonal form.

Arguments

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the diagonal elements of the tridiagonal matrix.

On exit, the computed eigenvalues replace the first m elements of D in decreasing sequence. The other elements are lost.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the tridiagonal matrix. E(n) is arbitrary and is used as workspace.

On exit, E has been destroyed.

The size of E must verify: size(E) = size(D) = n .

M (INPUT) integer(i4b) On entry, the number of smallest or largest eigenvalues wanted. M must be less than or equal to size(E) = size(D) = n.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the rational QR iterations for some eigenvalues.

SMALL (INPUT, OPTIONAL) logical(lgl) On entry:

- SMALL = false : indicates that the M largest eigenvalues are desired.
- SMALL = true : indicates that the M smallest eigenvalues are desired.

The default is false.

TOL (INPUT, OPTIONAL) real(stnd) On entry, TOL specifies a tolerance for the theoretical error of the computed eigenvalues. The theoretical error of the k-th eigenvalue is usually not greater than k * TOL.

The default is zero.

Further Details

This subroutine is not parallelized.

For further details, see:

(1) **Reinsch, C., and Bauer, F.L., 1968:** Rational QR transformation with Newton shift for symmetric tridiagonal matrices. Numerische Mathematik 11, 264-272.

6.4.25 `subroutine symtrid_ratqri2 ( d, e, val, failure, m, small, tol )`

Purpose

SYMTRID_RATQRI2 computes the largest or smallest eigenvalues of a symmetric n-by-n tridiagonal matrix whose sum of absolute values exceeds a given (positive) value. A rational QR method is used.

The largest or smallest eigenvalues of a full symmetric matrix whose sum of absolute values exceeds a given threshold can also be found, if SYMTRID_CMP has been used to reduce this matrix to tridiagonal form.

Arguments

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the diagonal elements of the tridiagonal matrix.

On exit, the computed eigenvalues replace the first M elements of D in decreasing sequence. The other elements are lost.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the tridiagonal matrix. E(n) is arbitrary and is used as workspace.

On exit, E has been destroyed.

The size of E must verify: size(E) = size(D) = n .

VAL (INPUT) real(stnd) On entry, the sum of the M eigenvalues (in absolute value) found will exceed abs(VAL) or M is equal to n.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the rational QR iterations for some eigenvalues.

M (OUTPUT) integer(i4b) On exit, the number of eigenvalues found.

SMALL (INPUT, OPTIONAL) logical(lgl) On entry:

- SMALL = false : indicates that the M largest eigenvalues are desired.
- SMALL = true : indicates that the M smallest eigenvalues are desired.

The default is false.

TOL (INPUT, OPTIONAL) real(stnd) On entry, TOL specifies a tolerance for the theoretical error of the computed eigenvalues. The theoretical error of the k-th eigenvalue is usually not greater than k * TOL.

The default is zero.

Further Details

This subroutine is not parallelized.

For further details, see:

(1) **Reinsch, C., and Bauer, F.L., 1968:** Rational QR transformation with Newton shift for symmetric tridiagonal matrices. Numerische Mathematik 11, 264-272.

6.4.26 `subroutine symtrid_bisect ( d, e, neig, w, failure, small, sort, vector, abstol, &`

Purpose

SYMTRID_BISECT computes all or some of the largest or smallest eigenvalues of a real n-by-n symmetric tridiagonal matrix T using a bisection method.

The largest or smallest eigenvalues of a full symmetric matrix can also be found if SYMTRID_CMP has been used to reduce this matrix to tridiagonal form.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the tridiagonal matrix T.

E (INPUT) real(stnd), dimension(:) On entry, E contains the n-1 off-diagonal elements of the tridiagonal matrix T whose eigenvalues are desired. E(n) is arbitrary.

The size of E must verify: size(E) = size(D) = n .

NEIG (OUTPUT) integer(i4b) On output, NEIG specifies the number of eigenvalues which have been computed. Note that NEIG may be greater than the optional argument LE, if multiple eigenvalues at index LE make unique selection impossible.

If none of the optional arguments LE and THETA are used, NEIG is set to size(D) and all the eigenvalues of T are computed.

W (OUTPUT) real(stnd), dimension(:) On exit, W(1:NEIG) contains the first NEIG largest (or smallest) eigenvalues of T. The other values in W (e.g. W(NEIG+1:size(D))) are flagged by a quiet NAN.

The size of W must verify: size(W) = size(D) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit and the bisection algorithm converged for all the computed eigenvalues to the desired accuracy;
- FAILURE = true : indicates that some or all of the eigenvalues failed to converge or were not computed. This is generally caused by unexpectedly inaccurate arithmetic.

SMALL (INPUT, OPTIONAL) logical(lgl) On entry:

- SMALL = false : indicates that the largest eigenvalues are desired;
- SMALL = true : indicates that the smallest eigenvalues are desired.

The default is false.

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. For other values of SORT nothing is done and W(:NEIG) may not be sorted.

VECTOR (INPUT, OPTIONAL) logical(lgl) On entry, if VECTOR is set to true, a vectorized version of the bisection algorithm is used.

The default is VECTOR=false.

ABSTOL (INPUT, OPTIONAL) real(stnd) On entry, the absolute tolerance for the eigenvalues. An eigenvalue (or cluster) is considered to be located if it has been determined to lie in an interval whose width is ABSTOL or less.

If ABSTOL is less than or equal to zero, or is not specified, then ULP * | T | will be used, where | T | means the 1-norm of T and ULP is the machine precision (distance from 1 to the next larger floating point number).

Eigenvalues will be computed most accurately when ABSTOL is set to the square root of the underflow threshold, sqrt(LAMCH('S')), not zero.

LE (INPUT, OPTIONAL) integer(i4b) On entry, LE specifies the number of eigenvalues which must be computed by the subroutine. On output, NEIG may be different than LE if multiple eigenvalues at index LE make unique selection impossible.

If:

- SMALL=false, the subroutine computes the LE largest eigenvalues of T ;
- SMALL=true, the subroutine computes the LE smallest eigenvalues of T .

Only one of the optional arguments LE and THETA must be specified, otherwise the subroutine will stop with an error message.

LE must be greater than 0 and less or equal to size(D) .

The default is LE = size(D).

THETA (INPUT, OPTIONAL) real(stnd) On entry:

- if SMALL=false, THETA specifies that the eigenvalues which are greater or equal to THETA must be computed. If none of the eigenvalues are greater or equal to THETA, NEIG is set to zero and W(:) to a quiet NAN;

- if SMALL=true, THETA specifies that the eigenvalues which are less or equal to THETA must be computed. If none of the eigenvalues are smaller or equal to THETA, NEIG is set to zero and W(:) to a quiet NAN.

Only one of the optional arguments LE and THETA must be specified, otherwise the subroutine will stop with an error message.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the tridiagonal matrix T is scaled before computing the eigenvalues.

The default is to scale the tridiagonal matrix.

INIT (INPUT, OPTIONAL) logical(lgl) On entry, if INIT=true the initial intervals for the bisection steps are computed from estimates of the eigenvalues of the tridiagonal matrix obtained from the Pal-Walker-Kahan algorithm.

The default is not to use the Pal-Walker-Kahan algorithm.

Further Details

Let S(i), i=1,. . . ,N=size(D), be the N eigenvalues of the symmetric tridiagonal matrix T in decreasing order of magnitude. SYMTRID_BISECT then computes the LE largest or smallest eigenvalues (or the eigenvalues which are greater/smaller or equal to THETA) of T by a bisection method (see the reference (1) below, Sec.8.5).

This subroutine is parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.4.27 `subroutine dflgen ( d, e, lambda, cs, sn )`

Purpose

DFLGEN computes deflation parameters (e.g. a chain of Givens rotations) for a symmetric unreduced n-by-n tridiagonal matrix T and a given eigenvalue of T.

On output, the arguments CS and SN contain, respectively, the vectors of the cosines and sines coefficients of the chain of n-1 planar rotations that deflates the real n-by-n symmetric tridiagonal matrix T corresponding to an eigenvalue LAMBDA.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the tridiagonal matrix.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the tridiagonal matrix.

The size of E must verify: size(E) = size(D) - 1.

LAMBDA (INPUT) real(stnd) On entry, an eigenvalue of the tridiagonal matrix.

CS (OUTPUT) real(stnd), dimension(:) On exit, the vector of the cosines coefficients of the chain of n-1 Givens rotations that deflates the symmetric tridiagonal matrix.

The size of CS must verify: size(CS) = size(E) = size(D) - 1.

SN (OUTPUT) real(stnd), dimension(:) On exit, the vector of the sines coefficients of the chain of n-1 Givens rotations that deflates the symmetric tridiagonal matrix.

The size of SN must verify: size(SN) = size(E) = size(D) - 1.

Further Details

This subroutine is adapted from the matlab routine DFLGEN given in the reference (1). No check is done in the subroutine to verify that the input tridiagonal matrix is unreduced.

For further details, see:

(1) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

6.4.28 `subroutine dflgen2 ( d, e, lambda, cs, sn, deflate )`

Purpose

DFLGEN2 computes and applies deflation parameters (e.g. a chain of Givens rotations) for a symmetric unreduced n-by-n tridiagonal matrix T and a given eigenvalue of T.

On output:

the arguments D and E contain, respectively, the new main diagonal and subdiagonal of the deflated symmetric tridiagonal matrix if DEFLATE is set to true.

the arguments CS and SN contain, respectively, the vectors of the cosines and sines coefficients of the chain of n-1 planar rotations that deflates the real n-by-n symmetric tridiagonal matrix T corresponding to the eigenvalue LAMBDA.

Arguments

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the diagonal elements of the tridiagonal matrix.

On exit, the new main diagonal of the symmetric tridiagonal matrix if DEFLATE=true. Otherwise, D is not changed.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the tridiagonal matrix.

On exit, the new subdiagonal of the symmetric tridiagonal matrix if DEFLATE=true. Otherwise, E is not changed.

The size of E must verify: size(E) = size(D) - 1.

LAMBDA (INPUT) real(stnd) On entry, an eigenvalue of the tridiagonal matrix.

CS (OUTPUT) real(stnd), dimension(:) On exit, the vector of the cosines coefficients of the chain of n-1 Givens rotations that deflates the symmetric tridiagonal matrix.

The size of CS must verify: size(CS) = size(E) = size(D) - 1.

SN (OUTPUT) real(stnd), dimension(:) On exit, the vector of the sines coefficients of the chain of n-1 Givens rotations that deflates the symmetric tridiagonal matrix.

The size of SN must verify: size(SN) = size(E) = size(D) - 1.

DEFLATE (OUTPUT) logical(lgl) On exit:

- DEFLATE = true : indicates successful exit.
- DEFLATE = false: indicates that full accuracy was not attained in the deflation of the tridiagonal matrix.

Further Details

This subroutine is adapted from the matlab routine DFLGEN given in the reference (1). No check is done in the subroutine to verify that the input tridiagonal matrix is unreduced.

For further details, see:

(1) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

6.4.29 `subroutine dflapp ( d, e, cs, sn, deflate )`

Purpose

DFLAPP deflates a real symmetric n-by-n tridiagonal matrix T by a chain of planar rotations produced by DFLGEN.

On output, the arguments D and E contain, respectively, the new main diagonal and subdiagonal of the deflated symmetric tridiagonal matrix if DEFLATE is set to true.

Arguments

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the diagonal elements of the tridiagonal matrix.

On exit, the new main diagonal of the symmetric tridiagonal matrix if DEFLATE=true. Otherwise, D is not changed.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the tridiagonal matrix.

On exit, the new subdiagonal of the symmetric tridiagonal matrix if DEFLATE=true. Otherwise, E is not changed.

The size of E must verify: size(E) = size(D) - 1.

CS (INPUT) real(stnd), dimension(:) On entry, the vector of the cosines coefficients of the chain of n-1 Givens rotations that deflates the symmetric tridiagonal matrix as computed by DFLGEN subroutine.

The size of CS must verify: size(CS) = size(E) = size(D) - 1.

SN (INPUT) real(stnd), dimension(:) On entry, the vector of the sines coefficients of the chain of n-1 Givens rotations that deflates the symmetric tridiagonal matrix as computed by DFLGEN subroutine.

The size of SN must verify: size(SN) = size(E) = size(D) - 1.

DEFLATE (OUTPUT) logical(lgl) On exit:

- DEFLATE = true : indicates successful exit.
- DEFLATE = false: indicates that full accuracy was not attained in the deflation of the tridiagonal matrix.

Further Details

This subroutine is adapted from the matlab routine DFLAPP given in the reference (1).

For further details, see:

(1) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

6.4.30 `subroutine qrstep ( d, e, lambda, cs, sn, deflate )`

Purpose

QRSTEP performs one QR step with a given shift LAMBDA on a n-by-n real symmetric unreduced tridiagonal matrix T.

On output, the arguments D and E contain, respectively, the new main diagonal and subdiagonal of the deflated symmetric tridiagonal. The chain of n-1 planar rotations produced during the QR step are saved in the arguments CS and SN.

Arguments

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the diagonal elements of the tridiagonal matrix.

On exit, the new main diagonal of the symmetric tridiagonal matrix after the QR step.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the tridiagonal matrix.

On exit, the new subdiagonal of the symmetric tridiagonal matrix after the QR step.

The size of E must verify: size(E) = size(D) - 1.

LAMBDA (INPUT) real(stnd) On entry, the shift used in the current QR step.

CS (OUTPUT) real(stnd), dimension(:) On exit, the vector of the cosines coefficients of the chain of n-1 Givens rotations for the current QR step.

The size of CS must verify: size(CS) = size(E) = size(D) - 1.

SN (OUTPUT) real(stnd), dimension(:) On exit, the vector of the sines coefficients of the chain of n-1 Givens rotations for the current QR step.

The size of SN must verify: size(SN) = size(E) = size(D) - 1.

DEFLATE (OUTPUT) logical(lgl) On exit:

- DEFLATE = true : indicates that deflation occured at the end of the step.
- DEFLATE = false: indicates that the last subdiagonal element of the tridiagonal matrix is not small.

Further Details

This subroutine is adapted from the matlab routine QRSTEP given in the reference (1). No check is done in the subroutine to verify that the input tridiagonal matrix is unreduced.

For further details, see:

(1) **Mastronardi, M., Van Barel, M., Van Camp, E., and Vandebril, R., 2006:** On computing the eigenvectors of a class of structured matrices. Journal of Computational and Applied Mathematics, 189, 580-591.

6.4.31 `subroutine prodgiv ( cs, sn, x )`

Purpose

PRODGIV applies a chain of n-1 planar rotations produced by DFLGEN, DFLGEN2 or QRSTEP to a vector of length n.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the input vector of length n.

On exit, the product of the chain of the n-1 planar rotations by the input vector.

CS (INPUT) real(stnd), dimension(:) On entry, the vector of the cosines coefficients of the chain of n-1 Givens rotations as computed by DFLGEN or DFLGEN2 subroutines.

The size of CS must verify: size(CS) = size(X) - 1.

SN (INPUT) real(stnd), dimension(:) On entry, the vector of the sines coefficients of the chain of n-1 Givens rotations as computed by DFLGEN or DFLGEN2 subroutines.

The size of SN must verify: size(SN) = size(CS) = size(X) - 1.

Further Details

This subroutine is adapted from the matlab routine PRODGIV given in the reference (1).

For further details, see:

(1) **Mastronardi, M., Van Barel, M., Van Camp, E., and Vandebril, R., 2006:** On computing the eigenvectors of a class of structured matrices. Journal of Computational and Applied Mathematics, 189, 580-591.

6.4.32 `function prodgiv_eigvec ( cs, sn )`

Purpose

PRODGIV_EIGVEC computes an eigenvector of a n-by-n symmetric tridiagonal matrix from a chain of n-1 planar rotations produced by DFLGEN, DFLGEN2 or QRSTEP.

Arguments

CS (INPUT) real(stnd), dimension(:) On entry, the vector of the cosines coefficients of the chain of n-1 Givens rotations as computed by DFLGEN or DFLGEN2 subroutines.

SN (INPUT) real(stnd), dimension(:) On entry, the vector of the sines coefficients of the chain of n-1 Givens rotations as computed by DFLGEN or DFLGEN2 subroutines.

The size of SN must verify: size(SN) = size(CS) = n - 1.

Further Details

This subroutine is adapted from the matlab routine PRODGIV given in the reference (1).

For further details, see:

(1) **Mastronardi, M., Van Barel, M., Van Camp, E., and Vandebril, R., 2006:** On computing the eigenvectors of a class of structured matrices. Journal of Computational and Applied Mathematics, 189, 580-591.

6.4.33 `subroutine symtrid_deflate ( d, e, eigval, eigvec, failure, max_qr_steps )`

Purpose

SYMTRID_DEFLATE computes an eigenvector of a real symmetric tridiagonal matrix T corresponding to a specified eigenvalue, using a deflation technique.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the symmetric tridiagonal matrix T.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the symmetric tridiagonal matrix T.

The size of E must verify: size(E) = size(D) - 1 = n - 1.

EIGVAL (INPUT) real(stnd) On entry, an eigenvalue of the symmetric tridiagonal matrix.

EIGVEC (OUTPUT) real(stnd), dimension(:) On exit, the computed eigenvector associated with the eigenvalue EIGVAL.

The sise of EIGVEC must verify: size(EIGVEC) = size(D) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the deflation procedure of the tridiagonal matrix.

MAX_QR_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_QR_STEPS controls the maximum number of QR sweeps for deflating the tridiagonal matrix for a given eigenvalue. The algorithm fails to converge if the total number of QR sweeps exceeds MAX_QR_STEPS.

The default is 4.

Further Details

SYMTRID_DEFLATE may fail for some zero-diagonal tridiagonal matrices.

For further details, see:

(1) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

(2) **Mastronardi, M., Van Barel, M., Van Camp, E., and Vandebril, R., 2006:** On computing the eigenvectors of a class of structured matrices. Journal of Computational and Applied Mathematics, 189, 580-591.

6.4.34 subroutine symtrid_deflate (d, e, eigval, eigvec, failure, max_qr_steps)

Purpose

SYMTRID_DEFLATE computes eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, using a deflation technique.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the symmetric tridiagonal matrix T.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the symmetric tridiagonal matrix T.

The size of E must verify: size(E) = size(D) - 1 = n - 1.

EIGVAL (INPUT) real(stnd), dimension(:) On entry, selected eigenvalues of the symmetric tridiagonal matrix. The eigenvalues can be given in any order.

The size of EIGVAL must verify: size(EIGVAL) <= size(D) = n .

EIGVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed eigenvectors. The eigenvector associated with the eigenvalue EIGVAL(j) is stored in the j-th column of EIGVEC.

The shape of EIGVEC must verify:

- size(EIGVEC, 1) = size(D) = n ;
- size(EIGVEC, 2) = size(EIGVAL).

FAILURE (OUTPUT) logical(lgl), dimension(:) On exit:

- FAILURE(j) = false : indicates successful exit for the jth eigenvector.
- FAILURE(j) = true : indicates that the algorithm did not converge and full accuracy was not attained in the deflation procedure of the tridiagonal matrix for the jth eigenvector.

The size of FAILURE must verify: size(FAILURE) = size(EIGVAL) .

MAX_QR_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_QR_STEPS controls the maximum number of QR sweeps for deflating the tridiagonal matrix for a given eigenvalue. The algorithm fails to converge if the total number of QR sweeps for all eigenvalues exceeds MAX_QR_STEPS * size(EIGVAL).

The default is 4.

Further Details

SYMTRID_DEFLATE may fail if some the eigenvalues specified in parameter EIGVAL are nearly identical or for clusters of small eigenvalues or for some zero-diagonal tridiagonal matrices.

For further details, see:

(1) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

(2) **Mastronardi, M., Van Barel, M., Van Camp, E., and Vandebril, R., 2006:** On computing the eigenvectors of a class of structured matrices. Journal of Computational and Applied Mathematics, 189, 580-591.

6.4.35 subroutine trid_deflate (d, e, eigval, eigvec, failure, max_qr_steps, ortho, inviter)

Purpose

TRID_DEFLATE computes the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, using deflation techniques.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the symmetric tridiagonal matrix T.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the symmetric tridiagonal matrix T, E(n) is arbitrary and is not used .

The size of E must verify: size(E) = size(D) = n.

EIGVAL (INPUT) real(stnd), dimension(:) On entry, selected eigenvalues of the symmetric tridiagonal matrix. The eigenvalues must be given in decreasing order.

The size of EIGVAL must verify: size(EIGVAL) <= size(D) = n .

EIGVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed eigenvectors. The eigenvector associated with the eigenvalue EIGVAL(j) is stored in the j-th column of EIGVEC.

The shape of EIGVEC must verify:

- size(EIGVEC, 1) = size(D) = n ;
- size(EIGVEC, 2) = size(EIGVAL) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the deflation procedure of the tridiagonal matrix.

MAX_QR_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_QR_STEPS controls the maximum number of QR sweeps for deflating the tridiagonal matrix for a given eigenvalue. The algorithm fails to converge if the total number of QR sweeps for all eigenvalues exceeds MAX_QR_STEPS * size(EIGVAL).

The default is 4.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, the tridiagonal matrix is deflated sequentially for all the specified eigenvalues; this implies that the eigenvectors will be automatically orthogonal on exit.
- ORTHO=false, the tridiagonal matrix is deflated in parallel for the different clusters of eigenvalues or isolated eigenvalues; this implies that orthogonality is preserved inside each cluster, but not automatically between clusters.

The default is ORTHO=false.

INVITER (INPUT, OPTIONAL) logical(lgl) On entry, if INVITER=true eigenvectors corresponding to isolated eigenvalues are computed by inverse iteration instead of deflation.

The default is INVITER=true.

Further Details

TRID_DEFLATE uses an efficient and robust approach for the computation of (selected) eigenvectors of a tridiagonal matrix corresponding to (selected) eigenvalues by combining Fernando's method for the computation of eigenvectors with deflation procedures by Givens rotations (see the references (1), (2) and (3) below). QR iterations are also used as a back-up procedure if the deflation technique fails (see the reference (4)).

Optionally, eigenvectors corresponding to isolated eigenvalues may be also computed by inverse iteration on the tridiagonal matrix T. This is the default for eigenvectors associated with isolated eigenvalues since in this case inverse iteration is safe and faster than the deflation algorithms.

The computation of the eigenvectors is parallelized if OPENMP is used.

It is essential that eigenvalues given on entry of TRID_DEFLATE are computed to high relative accuracy. Subroutine SYMTRID_BISECT may be used for this purpose.

TRID_DEFLATE may fail if some the eigenvalues specified in parameter EIGVAL are nearly identical or for clusters of small eigenvalues or for some zero-diagonal matrices.

The deflation algorithms used in TRID_DEFLATE are competitive with the inverse iteration procedure implemented in TRID_INVITER.

For further details, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(2) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinsin's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

(3) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

(4) **Mastronardi, M., Van Barel, M., Van Camp, E., and Vandebril, R., 2006:** On computing the eigenvectors of a class of structured matrices. Journal of Computational and Applied Mathematics, 189, 580-591.

6.4.36 `subroutine trid_deflate ( d, e, eigval, eigvec, failure, mat, max_qr_steps, ortho, inviter )`

Purpose

TRID_DEFLATE computes the eigenvectors of a full real n-by-n symmetric matrix MAT corresponding to specified eigenvalues, using deflation techniques applied to a symmetric tridiagonal matrix T followed by a back-transformation procedure.

It is required that the original symmetric matrix MAT has been reduced to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

with a call to SYMTRID_CMP with parameter STORE_Q set to true, before calling TRID_DEFLATE.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the symmetric tridiagonal form T of MAT.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the symmetric tridiagonal form T of MAT. E(n) is arbitrary .

The size of E must verify: size(E) = size(D) = n .

EIGVAL (INPUT) real(stnd), dimension(:) On entry, selected eigenvalues of the symmetric matrix MAT. The eigenvalues must be given in decreasing order.

The size of EIGVAL must verify: size(EIGVAL) <= size(D) = n .

EIGVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed eigenvectors. The eigenvector associated with the eigenvalue EIGVAL(j) is stored in the j-th column of EIGVEC.

The shape of EIGVEC must verify:

- size(EIGVEC, 1) = size(D) = n ;
- size(EIGVEC, 2) = size(EIGVAL).

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the deflation procedure of the tridiagonal matrix.

MAT (INPUT) real(stnd), dimension(:,:) On entry, the vectors and the scalars which define the elementary reflectors used to reduce the full real n-by-n symmetric matrix MAT to symmetric tridiagonal form T, as returned by SYMTRID_CMP with STORE_Q=true, in its argument MAT. MAT is not modified by the routine.

Back-transformation is used to find the selected eigenvectors of the original matrix MAT and these eigenvectors are stored in argument EIGVEC.

The shape of MAT must verify:

- size(MAT, 1) = size(D) = n ;
- size(MAT, 2) = size(D) = n .

MAX_QR_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_QR_STEPS controls the maximum number of QR sweeps for deflating the tridiagonal matrix for a given eigenvalue. The algorithm fails to converge if the total number of QR sweeps for all eigenvalues exceeds MAX_QR_STEPS * size(EIGVAL).

The default is 4.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, the tridiagonal matrix is deflated sequentially for all the specified eigenvalues; this implies that the eigenvectors will be automatically orthogonal on exit.
- ORTHO=false, the tridiagonal matrix is deflated in parallel for the different clusters of eigenvalues or isolated eigenvalues; this implies that orthogonality is preserved inside each cluster, but not automatically between clusters.

The default is ORTHO=false.

INVITER (INPUT, OPTIONAL) logical(lgl) On entry, if INVITER=true eigenvectors corresponding to isolated eigenvalues are computed by inverse iteration instead of deflation.

The default is INVITER=true.

Further Details

TRID_DEFLATE uses an efficient and robust approach for the computation of (selected) eigenvectors of a tridiagonal matrix corresponding to (selected) eigenvalues by combining Fernando's method for the computation of eigenvectors with deflation procedures by Givens rotations (see the references (1), (2) and (3) below). QR iterations are also used as a back-up procedure if the deflation technique fails (see the reference (4)).

Optionally, eigenvectors corresponding to isolated eigenvalues may be also computed by inverse iteration on the tridiagonal matrix T. This is the default for eigenvectors associated with isolated eigenvalues since in this case inverse iteration is safe and faster than the deflation algorithms.

In a second step, the corresponding (selected) eigenvectors of the full real n-by-n symmetric matrix MAT are computed by a blocked back-transformation algorithm using the Householder transformations used to reduce the full real n-by-n symmetric matrix MAT to symmetric tridiagonal form T (see the references (5) and (6)).

The computation of the eigenvectors is parallelized if OPENMP is used.

It is essential that eigenvalues given on entry of TRID_DEFLATE are computed to high relative accuracy. Subroutine SYMTRID_BISECT may be used for this purpose.

TRID_DEFLATE may fail if some the eigenvalues specified in parameter EIGVAL are nearly identical or for clusters of small eigenvalues or for some zero-diagonal matrices.

The deflation algorithms used in TRID_DEFLATE are competitive with the inverse iteration procedure implemented in TRID_INVITER.

For further details on the deflation algorithm and the blocked backed-transformation algorithm, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, 1013-1034.

(2) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinsin's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

(3) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

(4) **Mastronardi, M., Van Barel, M., Van Camp, E., and Vandebril, R., 2006:** On computing the eigenvectors of a class of structured matrices. Journal of Computational and Applied Mathematics, 189, 580-591.

(5) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(6) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.4.37 subroutine trid_deflate (d, e, eigval, eigvec, failure, matp, max_qr_steps, ortho, inviter)

Purpose

TRID_DEFLATE computes the eigenvectors of a full real n-by-n symmetric matrix MAT, packed columnwise in a linear array MATP, corresponding to specified eigenvalues, using deflation techniques applied

to a symmetric tridiagonal matrix T followed by a back-transformation procedure.

It is required that the original packed symmetric matrix mat has been reduced to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

with a call to SYMTRID_CMP with parameter STORE_Q set to true, before calling TRID_DEFLATE.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the symmetric tridiagonal form T of MAT.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the symmetric tridiagonal form T of MAT. E(n) is arbitrary .

The size of E must verify: size(E) = size(D) = n .

EIGVAL (INPUT) real(stnd), dimension(:) On entry, selected eigenvalues of the symmetric matrix MAT. The eigenvalues must be given in decreasing order.

The size of EIGVAL must verify: size(EIGVAL) <= size(D) = n .

EIGVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed eigenvectors. The eigenvector associated with the eigenvalue EIGVAL(j) is stored in the j-th column of EIGVEC.

The shape of EIGVEC must verify:

- size(EIGVEC, 1) = size(D) = n ;
- size(EIGVEC, 2) = size(EIGVAL) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the deflation procedure of the tridiagonal matrix.

MATP (INPUT) real(stnd), dimension(:) On entry, the vectors and the scalars which define the elementary reflectors used to reduce the packed real n-by-n symmetric matrix MAT to symmetric tridiagonal form T, as returned by SYMTRID_CMP with STORE_Q=true, in its argument MATP. MATP is not modified by the routine.

Back-transformation is used to find the selected eigenvectors of the original matrix MAT and these eigenvectors are stored in argument EIGVEC.

The size of MATP must verify: size(MATP) = (n * (n+1)/2)

MAX_QR_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_QR_STEPS controls the maximum number of QR sweeps for deflating the tridiagonal matrix for a given eigenvalue. The algorithm fails to converge if the total number of QR sweeps for all eigenvalues exceeds MAX_QR_STEPS * size(EIGVAL).

The default is 4.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, the tridiagonal matrix is deflated sequentially for all the specified eigenvalues; this implies that the eigenvectors will be automatically orthogonal on exit.

- ORTHO=false, the tridiagonal matrix is deflated in parallel for the different clusters of eigenvalues or isolated eigenvalues; this implies that orthogonality is preserved inside each cluster, but not automatically between clusters.

The default is ORTHO=false.

INVITER (INPUT, OPTIONAL) logical(lgl) On entry, if INVITER=true eigenvectors corresponding to isolated eigenvalues are computed by inverse iteration instead of deflation.

The default is INVITER=true.

Further Details

TRID_DEFLATE uses an efficient and robust approach for the computation of (selected) eigenvectors of a tridiagonal matrix corresponding to (selected) eigenvalues by combining Fernando's method for the computation of eigenvectors with deflation procedures by Givens rotations (see the references (1), (2) and (3) below). QR iterations are also used as a back-up procedure if the deflation technique fails (see the reference (4)).

Optionally, eigenvectors corresponding to isolated eigenvalues may be also computed by inverse iteration on the tridiagonal matrix T. This is the default for eigenvectors associated with isolated eigenvalues since in this case inverse iteration is safe and faster than the deflation algorithms.

In a second step, the corresponding (selected) eigenvectors of the full real n-by-n symmetric matrix MAT are computed by a blocked back-transformation algorithm with the Householder transformations used to reduce the full real n-by-n symmetric matrix MAT to symmetric tridiagonal form T (see the references (5) and (6)). These Householder transformations must be packed in the linear array MATP (as returned by SYMTRID_CMP) on entry of TRID_DEFLATE.

The computation of the eigenvectors is parallelized if OPENMP is used.

It is essential that eigenvalues given on entry of TRID_DEFLATE are computed to high relative accuracy. Subroutine SYMTRID_BISECT may be used for this purpose.

TRID_DEFLATE may fail if some the eigenvalues specified in parameter EIGVAL are nearly identical or for clusters of small eigenvalues or for some zero-diagonal matrices.

The deflation algorithms used in TRID_DEFLATE are competitive with the inverse iteration procedure implemented in TRID_INVITER.

For further details, on the deflation algorithm and the blocked backed-transformation algorithm, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, 1013-1034.

(2) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinsin's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

(3) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

(4) **Mastronardi, M., Van Barel, M., Van Camp, E., and Vandebril, R., 2006:** On computing the eigenvectors of a class of structured matrices. Journal of Computational and Applied Mathematics, 189, 580-591.

(5) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(6) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.4.38 subroutine eig_cmp (mat, eigval, failure, upper, sort, maxiter)

Purpose

EIG_CMP computes all eigenvalues and eigenvectors of a n-by-n real symmetric matrix MAT.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT.

If:

- UPPER = true: the leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT.
- UPPER = false: the leading n-by-n lower triangular part of MAT contains the lower triangular part of the matrix MAT.

On exit, MAT contains the orthonormal eigenvectors of the matrix MAT.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the eigenvalues.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of an intermediate tridiagonal form T of MAT .

UPPER (INPUT) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER= true : Upper triangular is stored ;
- UPPER= false: Lower triangular is stored .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The eigenvectors are reordered accordingly.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of MAT.

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(EIGVAL). Convergence usually occurs in about 2 * size(EIGVAL) QR sweeps.

The default is 30.

Further Details

The matrix MAT is first transformed to tridiagonal form T, then the eigenvalues and the eigenvectors are computed by the QR implicit algorithm.

The computations of the eigenvectors are parallelized if OPENMP is used.

The EIG_CMP driver is a good choice for small symmetric matrices. However, it can be significantly slower than the EIG_CMP2 and EIG_CMP3 drivers for large symmetric matrices since the computation of the eigenvectors is not done with a level 3 BLAS algorithm in EIG_CMP.

For further details, see:

(1) **Parlett, B.N., 1998:** The Symmetric Eigenvalue Problem. Revised edition, SIAM, Philadelphia.

(2) **Greenbaum, A., and J. Dongarra, J., 1989:** Experiments with QR/QL Methods for the Symmetric Tridiagonal Eigenproblem. LAPACK Working Note No 17, November 1989.

(3) **Demmel, J., Heath, M.T., and Van Der Vorst, H., 1993:** Parallel numerical linear algebra. Acta Numerica 2, 111-197.

6.4.39 `subroutine eig_cmp ( mat, eigval, failure, sort, maxiter )`

Purpose

EIG_CMP computes all eigenvalues and eigenvectors of a n-by-n real symmetric matrix MAT.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT. The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT. The strictly n-by-n lower triangular part of MAT is not referenced.

On exit, MAT contains the orthonormal eigenvectors of the matrix MAT.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the eigenvalues.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of an intermediate tridiagonal form T of MAT .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The eigenvectors are reordered accordingly.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of MAT.

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(EIGVAL). Convergence usually occurs in about 2 * size(EIGVAL) QR sweeps.

The default is 30.

Further Details

The matrix MAT is first transformed to tridiagonal form T, then the eigenvalues and the eigenvectors are computed by the QR implicit algorithm.

A level 3 BLAS algorithm is used for computing the tridiagonal form T . Furthermore, the computations of the tridiagonal form T and the eigenvectors are parallelized if OPENMP is used.

The EIG_CMP driver is a good choice for small and medium symmetric matrices. However, it can be significantly slower than the EIG_CMP2 and EIG_CMP3 drivers for large symmetric matrices since the computation of the eigenvectors is not done with a level 3 BLAS algorithm in EIG_CMP.

For further details, see

(1) **Parlett, B.N., 1998:** The Symmetric Eigenvalue Problem. Revised edition, SIAM, Philadelphia.

(2) **Greenbaum, A., and J. Dongarra, J., 1989:** Experiments with QR/QL Methods for the Symmetric Tridiagonal Eigenproblem. LAPACK Working Note No 17, November 1989.

(3) **Demmel, J., Heath, M.T., and Van Der Vorst, H., 1993:** Parallel numerical linear algebra. Acta Numerica 2, 111-197.

6.4.40 subroutine eig_cmp2 (mat, eigval, failure, upper, sort, maxiter, max_francis_steps)

Purpose

EIG_CMP2 computes all eigenvalues and eigenvectors of a n-by-n real symmetric matrix MAT.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT.

If:

- UPPER = true: the leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT.
- UPPER = false: the leading n-by-n lower triangular part of MAT contains the lower triangular part of the matrix MAT.

On exit, MAT contains the orthonormal eigenvectors of the matrix MAT.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the eigenvalues.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of an intermediate tridiagonal form T of MAT .

UPPER (INPUT) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular is stored ;
- UPPER = false: Lower triangular is stored .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The eigenvectors are reordered accordingly.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of MAT .

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(EIGVAL). Convergence usually occurs in about 2 * size(EIGVAL) QR sweeps.

The default is 30.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the eigenvectors in the QR algorithm. MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of n and the integer parameter MAX_FRANCIS_STEPS_EIG specified in the module Select_Parameters.

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

The matrix MAT is first transformed to tridiagonal form T, then the eigenvalues are computed by the Pal-Walker-Kahan variant of the QR algorithm and the eigenvectors are computed with a perfect shift strategy. If this strategy fails some eigenvalue-eigenvector couplet, this couplet is computed by the QR implicit algorithm.

A level 3 BLAS algorithms is used for computing the eigenvectors of the tridiagonal form T. Furthermore, the computations of the eigenvectors are parallelized if OPENMP is used.

The EIG_CMP2 driver is usually significantly faster than the EIG_CMP driver for medium or large symmectric matrices, but can be slightly slower than EIG_CMP for small symmetric matrices.

For further details, see:

(1) **Parlett, B.N., 1998:** The Symmetric Eigenvalue Problem. Revised edition, SIAM, Philadelphia.

(2) **Greenbaum, A., and J. Dongarra, J., 1989:** Experiments with QR/QL Methods for the Symmetric Tridiagonal Eigenproblem. LAPACK Working Note No 17, November 1989.

(3) **Lang, B., 1998:** Using level 3 BLAS in rotation-based algorithms. Siam J. Sci. Comput., Vol. 19, 626-634.

(4) **Van Zee, F.G., van de Geijn, R., and Quintana-Orti, G., 2011:** Restructuring the QR Algorithm for High-Performance Application of Givens Rotations. FLAME Working Note 60. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-11-36.

6.4.41 subroutine eig_cmp2 (mat, eigval, failure, sort, maxiter, max_francis_steps)

Purpose

EIG_CMP2 computes all eigenvalues and eigenvectors of a n-by-n real symmetric matrix MAT.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT. The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT. The strictly n-by-n lower triangular part of MAT is not referenced.

On exit, MAT contains the orthonormal eigenvectors of the matrix MAT.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the eigenvalues.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of an intermediate tridiagonal form T of MAT .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The eigenvectors are reordered accordingly.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of MAT .

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(EIGVAL). Convergence usually occurs in about 2 * size(EIGVAL) QR sweeps.

The default is 30.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the eigenvectors in the QR algorithm. MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of n and the integer parameter MAX_FRANCIS_STEPS_EIG specified in the module Select_Parameters.

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

The matrix MAT is first transformed to tridiagonal form T, then the eigenvalues are computed by the Pal-Walker-Kahan variant of the QR algorithm and the eigenvectors are computed with a perfect shift strategy. If this strategy fails some eigenvalue-eigenvector couplet, this couplet is computed by the QR implicit algorithm.

Level 3 BLAS algorithms are used for computing both the tridiagonal form T and the eigenvectors. Furthermore, the computations of the tridiagonal form T and the eigenvectors are parallelized if OPENMP is used.

The EIG_CMP2 driver is usually significantly faster than the EIG_CMP driver for medium or large symmectric matrices, but can be slightly slower than EIG_CMP for small symmetric matrices.

For further details, see:

(1) **Parlett, B.N., 1998:** The Symmetric Eigenvalue Problem. Revised edition, SIAM, Philadelphia.

(2) **Greenbaum, A., and J. Dongarra, J., 1989:** Experiments with QR/QL Methods for the Symmetric Tridiagonal Eigenproblem. LAPACK Working Note No 17, November 1989.

(3) **Lang, B., 1998:** Using level 3 BLAS in rotation-based algorithms. Siam J. Sci. Comput., Vol. 19, 626-634.

(4) **Van Zee, F.G., van de Geijn, R., and Quintana-Orti, G., 2011:** Restructuring the QR Algorithm for High-Performance Application of Givens Rotations. FLAME Working Note 60. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-11-36.

6.4.42 subroutine eig_cmp3 (mat, eigval, failure, upper, sort, maxiter, max_francis_steps)

Purpose

EIG_CMP3 computes all eigenvalues and eigenvectors of a n-by-n real symmetric matrix MAT.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT.

If:

- UPPER = true: the leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT.
- UPPER = false: the leading n-by-n lower triangular part of MAT contains the lower triangular part of the matrix MAT.

On exit, MAT contains the orthonormal eigenvectors of the matrix MAT.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the eigenvalues.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of an intermediate tridiagonal form T of MAT .

UPPER (INPUT) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular is stored ;
- UPPER = false: Lower triangular is stored .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The eigenvectors are reordered accordingly.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of MAT .

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(EIGVAL). Convergence usually occurs in about 2 * size(EIGVAL) QR sweeps.

The default is 30.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the eigenvectors in the QR algorithm. MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of n and the integer parameter MAX_FRANCIS_STEPS_EIG specified in the module Select_Parameters.

Further Details

The matrix MAT is first transformed to tridiagonal form T, then the eigenvalues and the eigenvectors are computed by the QR implicit algorithm.

A level 3 BLAS algorithms is used for computing the eigenvectors of the tridiagonal form T. Furthermore, the computations of the eigenvectors are parallelized if OPENMP is used.

The EIG_CMP3 driver is usually significantly faster than the EIG_CMP driver for medium or large symmectric matrices, but can be slightly slower than EIG_CMP for small symmetric matrices. It is also only slightly less efficient than the EIG_CMP2 driver for large matrices, but may be more robust for few specific symmetric matrices.

For further details, see:

(1) **Parlett, B.N., 1998:** The Symmetric Eigenvalue Problem. Revised edition, SIAM, Philadelphia.

(2) **Lang, B., 1998:** Using level 3 BLAS in rotation-based algorithms. Siam J. Sci. Comput., Vol. 19, 626-634.

(3) **Van Zee, F.G., Van de Geijn, R., and Quintana-Orti, G., 2011:** Restructuring the QR Algorithm for High-Performance Application of Givens Rotations. FLAME Working Note 60. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-11-36.

6.4.43 `subroutine eig_cmp3 ( mat, eigval, failure, sort, maxiter, max_francis_steps )`

Purpose

EIG_CMP3 computes all eigenvalues and eigenvectors of a n-by-n real symmetric matrix MAT.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT. The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT. The strictly n-by-n lower triangular part of MAT is not referenced.

On exit, MAT contains the orthonormal eigenvectors of the matrix MAT.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the eigenvalues.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of an intermediate tridiagonal form T of MAT .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The eigenvectors are reordered accordingly.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of MAT .

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(EIGVAL). Convergence usually occurs in about 2 * size(EIGVAL) QR sweeps.

The default is 30.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the eigenvectors in the QR algorithm. MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of n and the integer parameter MAX_FRANCIS_STEPS_EIG specified in the module Select_Parameters.

Further Details

The matrix MAT is first transformed to tridiagonal form T, then the eigenvalues and the eigenvectors are computed by the QR implicit algorithm.

Level 3 BLAS algorithms are used for computing both the tridiagonal form T and the eigenvectors. Furthermore, the computations of the tridiagonal form T and the eigenvectors are parallelized if OPENMP is used.

The EIG_CMP3 driver is usually significantly faster than the EIG_CMP driver for medium or large symmectric matrices, but can be slightly slower than EIG_CMP for small symmetric matrices. It is also only slightly less efficient than the EIG_CMP2 driver for large matrices, but may be more robust for few specific symmetric matrices.

For further details, see:

(1) **Parlett, B.N., 1998:** The Symmetric Eigenvalue Problem. Revised edition, SIAM, Philadelphia.

(2) **Lang, B., 1998:** Using level 3 BLAS in rotation-based algorithms. Siam J. Sci. Comput., Vol. 19, 626-634.

(3) **Van Zee, F.G., Van de Geijn, R., and Quintana-Orti, G., 2011:** Restructuring the QR Algorithm for High-Performance Application of Givens Rotations. FLAME Working Note 60. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-11-36.

6.4.44 subroutine eig_cmp4 (mat, eigval, failure, maxiter_qr, maxiter_inviter, ortho, backward_sweep, scaling, initvec)

Purpose

EIG_CMP4 computes all eigenvalues and eigenvectors of a n-by-n real symmetric matrix MAT using the Pal-Walker-Kahan variant of the QR algorithm for the eigenvalues and the inverse iteration method for the eigenvectors.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT. The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT. The strictly n-by-n lower triangular part of MAT is not referenced.

On exit, MAT contains the orthonormal eigenvectors of the matrix MAT .

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the eigenvalues of MAT sorted in descending order. The eigenvectors in MAT are reordered accordingly.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithms did not converge and that full accuracy was not attained in the Schur decomposition of an intermediate tridiagonal form T of MAT for the eigenvalues or at the end of the inverse iterations for the eigenvectors.

MAXITER_QR (INPUT, OPTIONAL) integer(i4b) MAXITER_QR controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of MAT when computing the eigenvalues.

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER_QR * size(EIGVAL). Convergence usually occurs in about 2 * size(EIGVAL) QR sweeps.

The default is 30.

MAXITER_INVITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed for computing the eigenvectors.

By default, 2 inverse iterations are performed for all the eigenvectors of an intermediate tridiagonal form T of MAT .

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, all the eigenvectors of the intermediate tridiagonal form T are orthogonalized by the Modified Gram-Schmidt or QR algorithm;
- ORTHO=false, the eigenvectors of T are not orthogonalized by the Modified Gram-Schmidt or QR algorithm.

The default is to orthogonalize the eigenvectors of T only if the eigenvalues are not well-separated.

BACKWARD_SWEEP (INPUT, OPTIONAL) logical(lgl) On entry, if:

- BACKWARD_SWEEP=true and the eigenvectors of T are orthogonalized by the modified Gram-Schmidt algorithm, a backward sweep of the modified Gram-Schmidt algorithm is also performed;
- BACKWARD_SWEEP=false a backward sweep is not performed.

The default is not to perform a backward sweep of the modified Gram-Schmidt algorithm.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if:

- SCALING=true, the tridiagonal matrix T is scaled before computing the eigenvectors;
- SCALING=false, the tridiagonal matrix T is not scaled.

The default is to scale the tridiagonal matrix.

INITVEC (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITVEC=true, Fernando vectors are used to start the inverse iteration process for computing the eigenvectors of T ;
- INITVEC=false, random uniform starting vectors are used.

For unreduced tridiagonal matrices T , the default is to use Fernando starting vectors if the eigenvalues are well-separated and random uniform starting vectors otherwise. For reduced tridiagonal matrices T , the default is to use random uniform starting vectors.

Further Details

The matrix MAT is first transformed to tridiagonal form T, then the eigenvalues are computed by the Pal-Walker-Kahan variant of the QR algorithm and the eigenvectors of T are computed with an inverse iteration method and are, finally, back-transformed into those of MAT .

For further details on the Symmetric Eigenvalue Problem, Fernando's method or inverse iteration for computing eigenvectors of tridiagonal matrices or, finally, the blocked back-transformation algorithm, see:

(1) **Parlett, B.N., 1998:** The Symmetric Eigenvalue Problem. Revised edition, SIAM, Philadelphia.

(2) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(3) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

(4) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(5) **Bini, D.A., Gemignani, L., and Tisseur, F., 2005:** The Ehrlich-Aberth method for the nonsymmetric tridiagonal eigenvalue problem. SIAM J. Matrix Anal. Appl., 27, 153-175.

(6) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(7) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.4.45 subroutine eig_cmp5 (mat, neig, eigval, failure, small, le, theta, abstol, scaling_bisect, maxiter_inviter, ortho, backward_sweep, scaling_inviter, initvec)

Purpose

EIG_CMP5 computes all or, the largest or smallest, eigenvalues and associated eigenvectors of a n-by-n real symmetric matrix MAT using the bisection algorithm for the eigenvalues and the inverse iteration method for the eigenvectors.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT. The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT. The strictly n-by-n lower triangular part of MAT is not referenced.

On exit, the first NEIG columns of MAT contains the computed orthonormal eigenvectors of the matrix MAT .

The other columns in MAT (e.g., MAT(:,NEIG+1:)) are flagged by a quiet NAN.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

NEIG (OUTPUT) integer(i4b) On output, NEIG specifies the number of eigenvalues and eigenvectors which have been computed. Note that NEIG may be greater than the optional argument LE, if multiple eigenvalues at index LE make unique selection impossible.

If none of the optional arguments LE and THETA are used, NEIG is set to n and all the eigenvalues and eigenvectors of MAT are computed.

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, EIGVAL(1:NEIG) contains the first NEIG largest or smallest eigenvalues of MAT sorted in descending order.

The other values in EIGVAL (e.g. EIGVAL(NEIG+1:)) are flagged by a quiet NAN.

The eigenvectors in MAT are reordered accordingly.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit, meaning that the bisection and inverse iteration algorithms converged for all the computed eigenvalues and eigenvectors to the desired accuracy ;
- FAILURE = true : indicates that the algorithms did not converge, meaning that some or all of the eigenvalues failed to converge or were not computed, or, that full accuracy was not attained in computing eigenvectors of an intermediate tridiagonal form T of MAT by inverse iterations.

SMALL (INPUT, OPTIONAL) logical(lgl) On entry:

- SMALL = false : indicates that the m largest eigenvalues and associated eigenvectors are desired.
- SMALL = true : indicates that the m smallest eigenvalues and associated eigenvectors are desired.

LE (INPUT, OPTIONAL) integer(i4b) On entry, LE specifies the number of eigenvalues which must be computed by the subroutine. However, on output, NEIG may be different than LE if multiple eigenvalues at index LE make unique selection impossible.

If:

- SMALL=false, the subroutine computes the LE largest eigenvalues of MAT,
- SMALL=true, the subroutine computes the LE smallest eigenvalues of MAT.

Only one of the optional arguments LE and THETA must be specified, otherwise the subroutine will stop with an error message.

LE must be greater than 0 and less or equal to size(EIGVAL) .

The default is LE = size(EIGVAL), e.g., all the eigenvalues are computed.

THETA (INPUT, OPTIONAL) real(stnd) On entry:

- if SMALL=false, THETA specifies that the eigenvalues which are greater or equal to THETA must be computed. If none of the eigenvalues are greater or equal to THETA, NEIG is set to zero and EIGVAL(:) and MAT(:,:) to a quiet NAN.
- if SMALL=true, THETA specifies that the eigenvalues which are less or equal to THETA must be computed. If none of the eigenvalues are smaller or equal to THETA, NEIG is set to zero and EIGVAL(:) and MAT(:,:) to a quiet NAN.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

ABSTOL (INPUT, OPTIONAL) real(stnd) On entry, the absolute tolerance for the eigenvalues. An eigenvalue (or cluster) is considered to be located in the bisection algorithm if it has been determined to lie in an interval whose width is ABSTOL or less.

If ABSTOL is less than or equal to zero, or is not specified, then ULP * | T | will be used, where | T | means the 1-norm of T (T is the intermediate tridiagonal form of MAT) and ULP is the machine precision (distance from 1 to the next larger floating point number).

Eigenvalues will be computed most accurately when ABSTOL is set to the square root of the underflow threshold, e.g., sqrt(LAMCH('S')) or the parameter SAFMIN in the module Num_Constants, not zero.

SCALING_BISECT (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING_BISECT=true the intermediate tridiagonal matrix T is scaled before computing the eigenvalues by bisection.

The default is to scale the tridiagonal matrix.

MAXITER_INVITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed for computing the eigenvectors.

By default, 2 inverse iterations are performed for all the selected eigenvectors of an intermediate tridiagonal form T of MAT .

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, the selected eigenvectors of the intermediate tridiagonal form T are orthogonalized by the Modified Gram-Schmidt or QR algorithm;
- ORTHO=false, the selected eigenvectors of T are not orthogonalized by the Modified Gram-Schmidt or QR algorithm.

The default is to orthogonalize the selected eigenvectors of T only if the associated eigenvalues are not well-separated.

BACKWARD_SWEEP (INPUT, OPTIONAL) logical(lgl) On entry, if:

- BACKWARD_SWEEP=true and the selected eigenvectors of T are orthogonalized by the modified Gram-Schmidt algorithm, a backward sweep of the modified Gram-Schmidt algorithm is also performed;
- BACKWARD_SWEEP=false a backward sweep is not performed.

The default is not to perform a backward sweep of the modified Gram-Schmidt algorithm.

SCALING_INVITER (INPUT, OPTIONAL) logical(lgl) On entry, if:

- SCALING_INVITER=true, the tridiagonal matrix T is scaled before computing the selected eigenvectors;
- SCALING_INVITER=false, the tridiagonal matrix T is not scaled.

The default is to scale the tridiagonal matrix.

INITVEC (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITVEC=true, Fernando vectors are used to start the inverse iteration process for computing the selected eigenvectors of T ;
- INITVEC=false, random uniform starting vectors are used.

For unreduced tridiagonal matrices T , the default is to use Fernando starting vectors if the selected eigenvalues are well-separated and random uniform starting vectors otherwise. For reduced tridiagonal matrices T , the default is to use random uniform starting vectors.

Further Details

The matrix MAT is first transformed to tridiagonal form T, then the eigenvalues are computed by a bisection algorithm and the eigenvectors of T are computed with an inverse iteration method and are, finally, back-transformed into those of MAT .

For further details on the Symmetric Eigenvalue Problem, bisection, Fernando's method or inverse iteration for computing eigenvectors of tridiagonal matrices or, finally, the blocked back-transformation algorithm, see:

(1) **Parlett, B.N., 1998:** The Symmetric Eigenvalue Problem. Revised edition, SIAM, Philadelphia.

(2) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(3) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

(4) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(5) **Bini, D.A., Gemignani, L., and Tisseur, F., 2005:** The Ehrlich-Aberth method for the nonsymmetric tridiagonal eigenvalue problem. SIAM J. Matrix Anal. Appl., 27, 153-175.

(6) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(7) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.4.46 `function eigvalues ( mat, upper, sort, maxiter )`

Purpose

Function EIGVALUES computes all eigenvalues of a n-by-n real symmetric matrix MAT.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT.

If:

- UPPER = true: The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT.
- UPPER = false: The leading n-by-n lower triangular part of MAT contains the lower triangular part of the matrix MAT.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

UPPER (INPUT) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular is stored ;
- UPPER = false: Lower triangular is stored .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of MAT .

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(MAT,1). Convergence usually occurs in about 2 * size(MAT,1) QR sweeps.

The default is 30.

Further Details

This function is adapted from the routine DSYEV in LAPACK77 (version 3).

The matrix MAT is first transformed to tridiagonal form T, then the eigenvalues are computed by the Pal-Walker-Kahan variant of the QR algorithm. If the QR algorithm fails to converge EIGVALUES returns a n-vector filled with NAN() function.

For further details, see:

(1) **Parlett, B.N., 1998:** The Symmetric Eigenvalue Problem. Revised edition, SIAM, Philadelphia.

6.4.47 `function eigvalues ( mat, sort, maxiter )`

Purpose

Function EIGVALUES computes all eigenvalues of a n-by-n real symmetric matrix MAT.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT. The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT. The strictly n-by-n lower triangular part of MAT is not used.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of MAT .

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(MAT,1). Convergence usually occurs in about 2 * size(MAT,1) QR sweeps.

The default is 30.

Further Details

This function is adapted from the routine DSYEV in LAPACK77 (version 3).

The matrix MAT is first transformed to tridiagonal form T, then the eigenvalues are computed by the Pal-Walker-Kahan variant of the QR algorithm. If the QR algorithm fails to converge EIGVALUES returns a n-vector filled with NAN() function.

For further details, see:

(1) **Parlett, B.N., 1998:** The Symmetric Eigenvalue Problem. Revised edition, SIAM, Philadelphia.

6.4.48 subroutine eigval_cmp (mat, eigval, failure, upper, sort, maxiter, d_e)

Purpose

EIGVAL_CMP computes all eigenvalues of a n-by-n real symmetric matrix MAT.

The matrix MAT is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

,then the eigenvalues are computed by the Pal-Walker-Kahan variant of the QR algorithm.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT.

If:

- UPPER = true: the leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT.
- UPPER = false: the leading n-by-n lower triangular part of MAT contains the lower triangular part of the matrix MAT.

On exit:

- If UPPER = true and D_E is present : The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = false and D_E is present : The leading n-by-n lower triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = true and D_E is absent : The leading n-by-n upper triangular part of MAT is destroyed.
- If UPPER = false and D_E is absent : The leading n-by-n lower triangular part of MAT is destroyed.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the eigenvalues.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of the intermediate tridiagonal form T of MAT .

UPPER (INPUT) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular is stored ;
- UPPER = false: Lower triangular is stored .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of the intermediate tridiagonal form T of MAT .

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(EIGVAL). Convergence usually occurs in about 2 * size(EIGVAL) QR sweeps.

The default is 30.

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of MAT. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of MAT. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.49 subroutine eigval_cmp (mat, eigval, failure, sort, maxiter, d_e)

Purpose

EIGVAL_CMP computes all eigenvalues of a n-by-n real symmetric matrix MAT.

The matrix MAT is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

,then the eigenvalues are computed by the Pal-Walker-Kahan variant of the QR algorithm.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT. The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT. The strictly n-by-n lower triangular part of MAT is not referenced.

On exit:

- If D_E is present: The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If D_E is absent : The leading n-by-n upper triangular part of MAT is destroyed.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the eigenvalues.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of an intermediate tridiagonal form T of MAT .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of MAT .

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(EIGVAL). Convergence usually occurs in about 2 * size(EIGVAL) QR sweeps.

The default is 30.

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of MAT. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of MAT. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.50 subroutine eigval_cmp (matp, eigval, failure, upper, sort, maxiter, d_e)

Purpose

EIGVAL_CMP computes all eigenvalues of a n-by-n real symmetric matrix mat stored in packed form in a linear array MATP.

The matrix mat is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * mat * Q = T

,then the eigenvalues are computed by the Pal-Walker-Kahan variant of the QR algorithm.

Arguments

MATP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the upper or lower triangle of the symmetric matrix mat, packed column-wise in a linear array. The j-th column of mat is stored in the array MATP as follows:

- if UPPER = true, MATP(i + (j-1) * j/2) = mat(i,j) for 1<=i<=j;
- if UPPER = false, MATP(i + (j-1) * (2 * n-j)/2) = mat(i,j) for j<=i<=n.

On exit:

- If D_E is present : MATP is overwritten by the matrix Q as a product of elementary reflectors.
- If D_E is absent : MATP is destroyed.

The size of MATP must verify: size(MATP) = (n * (n+1)/2)

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the eigenvalues.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of an intermediate tridiagonal form T of mat .

UPPER (INPUT) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix mat is stored in the linear array MATP. If:

- UPPER = true : Upper triangle of mat is stored;
- UPPER = false: Lower triangle of mat is stored.

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of mat .

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(EIGVAL). Convergence usually occurs in about 2 * size(EIGVAL) QR sweeps.

The default is 30.

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of mat. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of mat. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.51 subroutine eigval_cmp (matp, eigval, failure, sort, maxiter, d_e)

Purpose

EIGVAL_CMP computes all eigenvalues of a n-by-n real symmetric matrix mat stored in packed form in a linear array MATP.

The matrix mat is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * mat * Q = T

,then the eigenvalues are computed by the Pal-Walker-Kahan variant of the QR algorithm.

Arguments

MATP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the upper triangle of the symmetric matrix mat, packed column-wise in a linear array. The j-th column of mat is stored in the array MATP as follows:

MATP(i + (j-1) * j/2) = mat(i,j) for 1<=i<=j;

On exit:

- If D_E is present : MATP is overwritten by the matrix Q as a product of elementary reflectors.
- If D_E is absent : MATP is destroyed.

The size of MATP must verify: size(MATP) = (n * (n+1)/2)

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the eigenvalues.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of an intermediate tridiagonal form T of mat .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of mat .

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(EIGVAL). Convergence usually occurs in about 2 * size(EIGVAL) QR sweeps.

The default is 30.

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of mat. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of mat. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.52 subroutine eigval_cmp2 (mat, eigval, failure, upper, sort, maxiter, d_e)

Purpose

EIGVAL_CMP2 computes all eigenvalues of a n-by-n real symmetric matrix MAT.

The matrix MAT is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

,then the eigenvalues are computed by the Pal-Walker-Kahan variant of the QR algorithm.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT.

If:

- UPPER = true: the leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT.
- UPPER = false: the leading n-by-n lower triangular part of MAT contains the lower triangular part of the matrix MAT.

On exit:

- If UPPER = true and D_E is present : The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = false and D_E is present : The leading n-by-n lower triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = true and D_E is absent : The leading n-by-n upper triangular part of MAT is destroyed.
- If UPPER = false and D_E is absent : The leading n-by-n lower triangular part of MAT is destroyed.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the eigenvalues.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of an intermediate tridiagonal form T of MAT .

UPPER (INPUT) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular is stored ;
- UPPER = false: Lower triangular is stored .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of MAT .

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(EIGVAL). Convergence usually occurs in about 2 * size(EIGVAL) QR sweeps.

The default is 30.

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of MAT. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of MAT. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.53 subroutine eigval_cmp2 (mat, eigval, failure, sort, maxiter, d_e)

Purpose

EIGVAL_CMP2 computes all eigenvalues of a n-by-n real symmetric matrix MAT.

The matrix MAT is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

,then the eigenvalues are computed by the Pal-Walker-Kahan variant of the QR algorithm.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT. The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT. The strictly n-by-n lower triangular part of MAT is not referenced.

On exit:

- If D_E is present: The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If D_E is absent : The leading n-by-n upper triangular part of MAT is destroyed.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the eigenvalues.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of an intermediate tridiagonal form T of MAT .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of MAT .

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(EIGVAL). Convergence usually occurs in about 2 * size(EIGVAL) QR sweeps.

The default is 30.

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of MAT. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of MAT. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.54 subroutine eigval_cmp2 (matp, eigval, failure, upper, sort, maxiter, d_e)

Purpose

EIGVAL_CMP2 computes all eigenvalues of a n-by-n real symmetric matrix mat stored in packed form in a linear array MATP.

The matrix mat is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * mat * Q = T

,then the eigenvalues are computed by the Pal-Walker-Kahan variant of the QR algorithm.

Arguments

MATP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the upper or lower triangle of the symmetric matrix mat , packed column-wise in a linear array. The j-th column of mat is stored in the array MATP as follows:

- if UPPER = true, MATP(i + (j-1) * j/2) = mat(i,j) for 1<=i<=j;
- if UPPER = false, MATP(i + (j-1) * (2 * n-j)/2) = mat(i,j) for j<=i<=n.

On exit:

- If D_E is present : MATP is overwritten by the matrix Q as a product of elementary reflectors.
- If D_E is absent : MATP is destroyed.

The size of MATP must verify: size(MATP) = (n * (n+1)/2)

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the eigenvalues.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of an intermediate tridiagonal form T of mat .

UPPER (INPUT) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix mat is stored in the linear array MATP. If:

- UPPER = true : Upper triangle of mat is stored;
- UPPER = false: Lower triangle of mat is stored.

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of mat .

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(EIGVAL). Convergence usually occurs in about 2 * size(EIGVAL) QR sweeps.

The default is 30.

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of mat. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of mat. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.55 subroutine eigval_cmp2 (matp, eigval, failure, sort, maxiter, d_e)

Purpose

EIGVAL_CMP2 computes all eigenvalues of a n-by-n real symmetric matrix mat stored in packed form in a linear array MATP.

The matrix mat is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * mat * Q = T

,then the eigenvalues are computed by the Pal-Walker-Kahan variant of the QR algorithm.

Arguments

MATP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the upper triangle of the symmetric matrix mat, packed column-wise in a linear array. The j-th column of mat is stored in the array MATP as follows:

MATP(i + (j-1) * j/2) = mat(i,j) for 1<=i<=j;

On exit:

- If D_E is present : MATP is overwritten by the matrix Q as a product of elementary reflectors.
- If D_E is absent : MATP is destroyed.

The size of MATP must verify: size(MATP) = (n * (n+1)/2)

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the eigenvalues.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of an intermediate tridiagonal form T of mat .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of mat .

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(EIGVAL). Convergence usually occurs in about 2 * size(EIGVAL) QR sweeps.

The default is 30.

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of mat. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of mat. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.56 subroutine eigval_cmp3 (mat, eigval, failure, upper, sort, maxiter, d_e)

Purpose

EIGVAL_CMP3 computes all eigenvalues of a n-by-n real symmetric matrix MAT.

The matrix MAT is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

,then the eigenvalues are computed by the implicit QR algorithm.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT.

If:

- UPPER = true: the leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT.
- UPPER = false: the leading n-by-n lower triangular part of MAT contains the lower triangular part of the matrix MAT.

On exit:

- If UPPER = true and D_E is present : The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = false and D_E is present : The leading n-by-n lower triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = true and D_E is absent : The leading n-by-n upper triangular part of MAT is destroyed.
- If UPPER = false and D_E is absent : The leading n-by-n lower triangular part of MAT is destroyed.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the eigenvalues.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of an intermediate tridiagonal form T of MAT .

UPPER (INPUT) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular is stored ;
- UPPER = false: Lower triangular is stored .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of MAT .

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(EIGVAL). Convergence usually occurs in about 2 * size(EIGVAL) QR sweeps.

The default is 30.

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of MAT. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of MAT. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.57 subroutine eigval_cmp3 (mat, eigval, failure, sort, maxiter, d_e)

Purpose

EIGVAL_CMP3 computes all eigenvalues of a n-by-n real symmetric matrix MAT.

The matrix MAT is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

,then the eigenvalues are computed by the implicit QR algorithm.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT. The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT. The strictly n-by-n lower triangular part of MAT is not referenced.

On exit:

- If D_E is present: The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If D_E is absent : The leading n-by-n upper triangular part of MAT is destroyed.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the eigenvalues.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of an intermediate tridiagonal form T of MAT .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of MAT .

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(EIGVAL). Convergence usually occurs in about 2 * size(EIGVAL) QR sweeps.

The default is 30.

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of MAT. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of MAT. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.58 subroutine eigval_cmp3 (matp, eigval, failure, upper, sort, maxiter, d_e)

Purpose

EIGVAL_CMP3 computes all eigenvalues of a n-by-n real symmetric matrix mat stored in packed form in a linear array MATP.

The matrix mat is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * mat * Q = T

,then the eigenvalues are computed by the implicit QR algorithm.

Arguments

MATP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the upper or lower triangle of the symmetric matrix mat , packed column-wise in a linear array. The j-th column of mat is stored in the array MATP as follows:

- if UPPER = true, MATP(i + (j-1) * j/2) = mat(i,j) for 1<=i<=j;
- if UPPER = false, MATP(i + (j-1) * (2 * n-j)/2) = mat(i,j) for j<=i<=n.

On exit:

- If D_E is present : MATP is overwritten by the matrix Q as a product of elementary reflectors.
- If D_E is absent : MATP is destroyed.

The size of MATP must verify: size(MATP) = (n * (n+1)/2)

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the eigenvalues.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of an intermediate tridiagonal form T of mat .

UPPER (INPUT) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix mat is stored in the linear array MATP. If:

- UPPER = true : Upper triangle of mat is stored ;
- UPPER = false: Lower triangle of mat is stored .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of mat .

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(EIGVAL). Convergence usually occurs in about 2 * size(EIGVAL) QR sweeps.

The default is 30.

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of mat. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of mat. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.59 subroutine eigval_cmp3 (matp, eigval, failure, sort, maxiter, d_e)

Purpose

EIGVAL_CMP3 computes all eigenvalues of a n-by-n real symmetric matrix mat stored in packed form in a linear array MATP.

The matrix mat is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * mat * Q = T

,then the eigenvalues are computed by the implicit QR algorithm.

Arguments

MATP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the upper triangle of the symmetric matrix mat , packed column-wise in a linear array. The j-th column of mat is stored in the array MATP as follows:

MATP(i + (j-1) * j/2) = mat(i,j) for 1<=i<=j;

On exit:

- If D_E is present : MATP is overwritten by the matrix Q as a product of elementary reflectors.
- If D_E is absent : MATP is destroyed.

The size of MATP must verify: size(MATP) = (n * (n+1)/2)

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the eigenvalues.

The size of EIGVAL must verify: size(EIGVAL) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the Schur decomposition of an intermediate tridiagonal form T of mat .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of mat .

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(EIGVAL). Convergence usually occurs in about 2 * size(EIGVAL) QR sweeps.

The default is 30.

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2)

On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of mat. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of mat. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.60 subroutine select_eigval_cmp (mat, eigval, small, failure, upper, d_e)

Purpose

SELECT_EIGVAL_CMP computes the m=size(EIGVAL) largest or smallest eigenvalues of a n-by-n real symmetric matrix MAT.

The matrix MAT is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

,then the eigenvalues are computed by a rational QR method.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT.

If:

- UPPER = true: the leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT.
- UPPER = false: the leading n-by-n lower triangular part of MAT contains the lower triangular part of the matrix MAT.

On exit:

- If UPPER = true and D_E is present : The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = false and D_E is present : The leading n-by-n lower triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = true and D_E is absent : The leading n-by-n upper triangular part of MAT is destroyed.
- If UPPER = false and D_E is absent : The leading n-by-n lower triangular part of MAT is destroyed.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the m=size(EIGVAL) largest or smallest eigenvalues of MAT in decreasing sequence.

The size of EIGVAL must verify: size(EIGVAL)<= size(MAT, 1) = size(MAT, 2) = n .

SMALL (INPUT) logical(lgl) On entry:

- SMALL = false : indicates that the m largest eigenvalues are desired.
- SMALL = true : indicates that the m smallest eigenvalues are desired.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the rational QR iterations for some eigenvalues of the intermediate tridiagonal form T of MAT .

UPPER (INPUT) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular is stored ;
- UPPER = false: Lower triangular is stored .

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of MAT. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of MAT. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.61 subroutine select_eigval_cmp (mat, eigval, small, failure, d_e)

Purpose

SELECT_EIGVAL_CMP computes the m=size(EIGVAL) largest or smallest eigenvalues of a n-by-n real symmetric matrix MAT.

The matrix MAT is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

,then the eigenvalues are computed by a rational QR method.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT. The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT. The strictly n-by-n lower triangular part of MAT is not referenced.

On exit:

- If D_E is present: The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If D_E is absent : The leading n-by-n upper triangular part of MAT is destroyed.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the m=size(EIGVAL) largest or smallest eigenvalues of MAT in decreasing sequence.

The size of EIGVAL must verify: size(EIGVAL)<= size(MAT, 1) = size(MAT, 2) = n .

SMALL (INPUT) logical(lgl) On entry:

- SMALL = false : indicates that the m largest eigenvalues are desired.
- SMALL = true : indicates that the m smallest eigenvalues are desired.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the rational QR iterations for some eigenvalues of the intermediate tridiagonal form T of MAT .

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of MAT. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of MAT. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.62 subroutine select_eigval_cmp (matp, eigval, small, failure, upper, d_e)

Purpose

SELECT_EIGVAL_CMP computes the m=size(EIGVAL) largest or smallest eigenvalues of a n-by-n real symmetric matrix mat stored in packed form in a linear array MATP.

The matrix mat is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * mat * Q = T

,then the eigenvalues are computed by a rational QR method.

Arguments

MATP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the upper or lower triangle of the symmetric matrix mat, packed column-wise in a linear array. The j-th column of mat is stored in the array MATP as follows:

- if UPPER = true, MATP(i + (j-1) * j/2) = mat(i,j) for 1<=i<=j;
- if UPPER = false, MATP(i + (j-1) * (2 * n-j)/2) = mat(i,j) for j<=i<=n.

On exit:

- If D_E is present : MATP is overwritten by the matrix Q as a product of elementary reflectors.
- If D_E is absent : MATP is destroyed.

The size of MATP must verify: size(MATP) = (n * (n+1)/2).

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the m=size(EIGVAL) largest or smallest eigenvalues of MAT in decreasing sequence.

The size of EIGVAL must verify: (m * (m+1))/2 <= size(MATP) = (n * (n+1)/2) .

SMALL (INPUT) logical(lgl) On entry:

- SMALL = false : indicates that the m largest eigenvalues are desired.
- SMALL = true : indicates that the m smallest eigenvalues are desired.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the rational QR iterations for some eigenvalues of the intermediate tridiagonal form T of MAT .

UPPER (INPUT) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix mat is stored in the linear array MATP. If:

- UPPER = true : Upper triangle of mat is stored;
- UPPER = false: Lower triangle of mat is stored.

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of mat. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of mat. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.63 subroutine select_eigval_cmp (matp, eigval, small, failure, d_e)

Purpose

SELECT_EIGVAL_CMP computes the m=size(EIGVAL) largest or smallest eigenvalues of a n-by-n real symmetric matrix mat stored in packed form in a linear array MATP.

The matrix mat is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * mat * Q = T

,then the eigenvalues are computed by a rational QR method.

Arguments

MATP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the upper triangle of the symmetric matrix mat, packed column-wise in a linear array. The j-th column of mat is stored in the array MATP as follows:

MATP(i + (j-1) * j/2) = mat(i,j) for 1<=i<=j;

On exit:

- If D_E is present : MATP is overwritten by the matrix Q as a product of elementary reflectors.
- If D_E is absent : MATP is destroyed.

The size of MATP must verify: size(MATP) = (n * (n+1)/2).

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, the m=size(EIGVAL) largest or smallest eigenvalues of MAT in decreasing sequence.

The size of EIGVAL must verify: (m * (m+1))/2 <= size(MATP) = (n * (n+1)/2) .

SMALL (INPUT) logical(lgl) On entry:

- SMALL = false : indicates that the m largest eigenvalues are desired.
- SMALL = true : indicates that the m smallest eigenvalues are desired.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the rational QR iterations for some eigenvalues of the intermediate tridiagonal form T of MAT .

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of mat. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of mat. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.64 subroutine select_eigval_cmp2 (mat, eigval, small, val, failure, upper, d_e)

Purpose

SELECT_EIGVAL_CMP2 computes the largest or smallest eigenvalues of a n-by-n real symmetric matrix MAT whose sum of absolute values exceeds a given (positive) threshold.

The matrix MAT is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

,then the eigenvalues are computed by a rational QR method.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT.

If:

- UPPER = true: the leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT.
- UPPER = false: the leading n-by-n lower triangular part of MAT contains the lower triangular part of the matrix MAT.

On exit:

- If UPPER = true and D_E is present : The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = false and D_E is present : The leading n-by-n lower triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = true and D_E is absent : The leading n-by-n upper triangular part of MAT is destroyed.
- If UPPER = false and D_E is absent : The leading n-by-n lower triangular part of MAT is destroyed.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

EIGVAL (OUTPUT) real(stnd), pointer, dimension(:) On exit, the computed largest or smallest eigenvalues of MAT in decreasing sequence.

On exit, m = size(EIGVAL) gives the number of computed eigenvalues.

SMALL (INPUT) logical(lgl) On entry:

- SMALL = false : indicates that the largest eigenvalues are desired.
- SMALL = true : indicates that the smallest eigenvalues are desired.

VAL (INPUT) real(stnd) On entry, the sum of the m eigenvalues (in absolute value) found will exceed abs(VAL) or m is equal to n.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the rational QR iterations for some eigenvalues of the intermediate tridiagonal form T of MAT .

UPPER (INPUT) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular is stored ;
- UPPER = false: Lower triangular is stored .

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of MAT. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of MAT. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.65 subroutine select_eigval_cmp2 (mat, eigval, small, val, failure, d_e)

Purpose

SELECT_EIGVAL_CMP2 computes the largest or smallest eigenvalues of a n-by-n real symmetric matrix MAT whose sum of absolute values exceeds a given (positive) threshold.

The matrix MAT is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

,then the eigenvalues are computed by a rational QR method.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT. The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT. The strictly n-by-n lower triangular part of MAT is not referenced.

On exit:

- If D_E is present: The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If D_E is absent : The leading n-by-n upper triangular part of MAT is destroyed.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

EIGVAL (OUTPUT) real(stnd), pointer, dimension(:) On exit, the computed largest or smallest eigenvalues of MAT in decreasing sequence.

On exit, m = size(EIGVAL) gives the number of computed eigenvalues.

SMALL (INPUT) logical(lgl) On entry:

- SMALL = false : indicates that the largest eigenvalues are desired.
- SMALL = true : indicates that the smallest eigenvalues are desired.

VAL (INPUT) real(stnd) On entry, the sum of the m eigenvalues (in absolute value) found will exceed abs(VAL) or m is equal to n.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the rational QR iterations for some eigenvalues of the intermediate tridiagonal form T of MAT .

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of MAT. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of MAT. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.66 subroutine select_eigval_cmp2 (matp, eigval, small, val, failure, upper, d_e)

Purpose

SELECT_EIGVAL_CMP2 computes the largest or smallest eigenvalues of a n-by-n real symmetric matrix mat, stored in packed form in a linear array MATP, whose sum of absolute values exceeds a given (positive) threshold.

The matrix mat is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * mat * Q = T

,then the eigenvalues are computed by a rational QR method.

Arguments

MATP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the upper or lower triangle of the symmetric matrix mat, packed column-wise in a linear array. The j-th column of mat is stored in the array MATP as follows:

- if UPPER = true, MATP(i + (j-1) * j/2) = mat(i,j) for 1<=i<=j;
- if UPPER = false, MATP(i + (j-1) * (2 * n-j)/2) = mat(i,j) for j<=i<=n.

On exit:

- If D_E is present : MATP is overwritten by the matrix Q as a product of elementary reflectors.
- If D_E is absent : MATP is destroyed.

The size of MATP must verify: size(MATP) = (n * (n+1)/2).

EIGVAL (OUTPUT) real(stnd), pointer, dimension(:) On exit, the computed largest or smallest eigenvalues of MAT in decreasing sequence.

On exit, m = size(EIGVAL) gives the number of computed eigenvalues.

SMALL (INPUT) logical(lgl) On entry:

- SMALL = false : indicates that the largest eigenvalues are desired.
- SMALL = true : indicates that the smallest eigenvalues are desired.

VAL (INPUT) real(stnd) On entry, the sum of the m eigenvalues (in absolute value) found will exceed abs(VAL) or m is equal to n.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the rational QR iterations for some eigenvalues of the intermediate tridiagonal form T of MAT .

UPPER (INPUT) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix mat is stored in the linear array MATP. If:

- UPPER = true : Upper triangle of mat is stored;
- UPPER = false: Lower triangle of mat is stored.

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of mat. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of mat. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.67 subroutine select_eigval_cmp2 (matp, eigval, small, val, failure, d_e)

Purpose

SELECT_EIGVAL_CMP2 computes the largest or smallest eigenvalues of a n-by-n real symmetric matrix mat, stored in packed form in a linear array MATP, whose sum of absolute values exceeds a given (positive) threshold.

The matrix mat is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * mat * Q = T

,then the eigenvalues are computed by a rational QR method.

Arguments

MATP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the upper triangle of the symmetric matrix mat , packed column-wise in a linear array. The j-th column of mat is stored in the array MATP as follows:

MATP(i + (j-1) * j/2) = mat(i,j) for 1<=i<=j;

On exit:

- If D_E is present : MATP is overwritten by the matrix Q as a product of elementary reflectors.
- If D_E is absent : MATP is destroyed.

The size of MATP must verify: size(MATP) = (n * (n+1)/2).

EIGVAL (OUTPUT) real(stnd), pointer, dimension(:) On exit, the computed largest or smallest eigenvalues of MAT in decreasing sequence.

On exit, m = size(EIGVAL) gives the number of computed eigenvalues.

SMALL (INPUT) logical(lgl) On entry:

- SMALL = false : indicates that the largest eigenvalues are desired.
- SMALL = true : indicates that the smallest eigenvalues are desired.

VAL (INPUT) real(stnd) On entry, the sum of the m eigenvalues (in absolute value) found will exceed abs(VAL) or m is equal to n.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the rational QR iterations for some eigenvalues of the intermediate tridiagonal form T of MAT .

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of mat. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of mat. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.68 subroutine select_eigval_cmp3 (mat, neig, eigval, small, failure, upper, sort, vector, scaling, init, abstol, le, theta, d_e)

Purpose

SELECT_EIGVAL_CMP3 computes the largest or smallest eigenvalues of a n-by-n real symmetric matrix MAT.

The matrix MAT is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

,then the eigenvalues are computed by a bisection method.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT.

If:

- UPPER = true: the leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT.
- UPPER = false: the leading n-by-n lower triangular part of MAT contains the lower triangular part of the matrix MAT.

On exit:

- If UPPER = true and D_E is present : The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = false and D_E is present : The leading n-by-n lower triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If UPPER = true and D_E is absent : The leading n-by-n upper triangular part of MAT is destroyed.
- If UPPER = false and D_E is absent : The leading n-by-n lower triangular part of MAT is destroyed.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

NEIG (OUTPUT) integer(i4b) On output, NEIG specifies the number of eigenvalues which have been computed. Note that NEIG may be greater than the optional argument LE, if multiple eigenvalues at index LE make unique selection impossible.

If none of the optional arguments LE and THETA are used, NEIG is set to n and all the eigenvalues of MAT are computed.

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, EIGVAL(1:NEIG) contains the first NEIG largest or smallest eigenvalues of MAT. The other values in EIGVAL (e.g. EIGVAL(NEIG+1:)) are flagged by a quiet NAN.

The size of EIGVAL must verify: size(EIGVAL) = size(MAT, 1) = size(MAT, 2) = n .

SMALL (INPUT) logical(lgl) On entry:

- SMALL = false : indicates that the largest eigenvalues are desired.
- SMALL = true : indicates that the smallest eigenvalues are desired.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit and the bisection algorithm converged for all the computed eigenvalues to the desired accuracy ;
- FAILURE = true : indicates that some or all of the eigenvalues failed to converge or were not computed. This is generally caused by unexpectedly inaccurate arithmetic.

UPPER (INPUT) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular is stored ;
- UPPER = false: Lower triangular is stored .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. For other values of SORT nothing is done and EIGVAL(:NEIG) may not be sorted.

VECTOR (INPUT, OPTIONAL) logical(lgl) On entry, if VECTOR is set to true, a vectorized version of the bisection algorithm is used to find the eigenvalues of the intermediate tridiagonal form T of MAT.

The default is VECTOR=false.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the intermediate tridiagonal matrix T is scaled before computing the eigenvalues.

The default is to scale the tridiagonal matrix.

INIT (INPUT, OPTIONAL) logical(lgl) On entry, if INIT=true the initial intervals for the bisection steps for computing the eigenvalues of the intermediate tridiagonal matrix T are estimated from the eigenvalues of the intermediate tridiagonal matrix obtained from the Pal-Walker-Kahan algorithm.

The default is not to use the Pal-Walker-Kahan algorithm.

ABSTOL (INPUT, OPTIONAL) real(stnd) On entry, the absolute tolerance for the eigenvalues. An eigenvalue (or cluster) is considered to be located if it has been determined to lie in an interval whose width is ABSTOL or less.

If ABSTOL is less than or equal to zero, or is not specified, then ULP * | T | will be used, where | T | means the 1-norm of T (T is the intermediate tridiagonal form of MAT) and ULP is the machine precision (distance from 1 to the next larger floating point number).

Eigenvalues will be computed most accurately when ABSTOL is set to the square root of the underflow threshold, sqrt(LAMCH('S')), not zero.

LE (INPUT, OPTIONAL) integer(i4b) On entry, LE specifies the number of eigenvalues which must be computed by the subroutine. However, on output, NEIG may be different than LE if multiple eigenvalues at index LE make unique selection impossible.

If:

- SMALL=false, the subroutine computes the LE largest eigenvalues of MAT,
- SMALL=true, the subroutine computes the LE smallest eigenvalues of MAT.

Only one of the optional arguments LE and THETA must be specified, otherwise the subroutine will stop with an error message.

LE must be greater than 0 and less or equal to size(EIGVAL) .

The default is LE = size(EIGVAL), e.g. all the eigenvalues are computed.

THETA (INPUT, OPTIONAL) real(stnd) On entry:

- if SMALL=false, THETA specifies that the eigenvalues which are greater or equal to THETA must be computed. If none of the eigenvalues are greater or equal to THETA, NEIG is set to zero and EIGVAL(:) to a quiet NAN.
- if SMALL=true, THETA specifies that the eigenvalues which are less or equal to THETA must be computed. If none of the eigenvalues are smaller or equal to THETA, NEIG is set to zero and EIGVAL(:) to a quiet NAN.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of MAT. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of MAT. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.69 subroutine select_eigval_cmp3 (mat, neig, eigval, small, failure, sort, vector, scaling, init, abstol, le, theta, d_e)

Purpose

SELECT_EIGVAL_CMP3 computes the largest or smallest eigenvalues of a n-by-n real symmetric matrix MAT.

The matrix MAT is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

,then the eigenvalues are computed by a bisection method.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT. The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix MAT. The strictly n-by-n lower triangular part of MAT is not referenced.

On exit:

- If D_E is present: The leading n-by-n upper triangular part of MAT is overwritten by the matrix Q as a product of elementary reflectors.
- If D_E is absent : The leading n-by-n upper triangular part of MAT is destroyed.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

NEIG (OUTPUT) integer(i4b) On output, NEIG specifies the number of eigenvalues which have been computed. Note that NEIG may be greater than the optional argument LE, if multiple eigenvalues at index LE make unique selection impossible.

If none of the optional arguments LE and THETA are used, NEIG is set to n and all the eigenvalues of MAT are computed.

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, EIGVAL(1:NEIG) contains the first NEIG largest or smallest eigenvalues of MAT. The other values in EIGVAL (e.g. EIGVAL(NEIG+1:)) are flagged by a quiet NAN.

The size of EIGVAL must verify: size(EIGVAL) = size(MAT, 1) = size(MAT, 2) = n .

SMALL (INPUT) logical(lgl) On entry:

- SMALL = false : indicates that the m largest eigenvalues are desired.
- SMALL = true : indicates that the m smallest eigenvalues are desired.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit and the bisection algorithm converged for all the computed eigenvalues to the desired accuracy ;
- FAILURE = true : indicates that some or all of the eigenvalues failed to converge or were not computed. This is generally caused by unexpectedly inaccurate arithmetic.

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. For other values of SORT nothing is done and EIGVAL(:NEIG) may not be sorted.

VECTOR (INPUT, OPTIONAL) logical(lgl) On entry, if VECTOR is set to true, a vectorized version of the bisection algorithm is used to find the eigenvalues of the intermediate tridiagonal form T of MAT.

The default is VECTOR=false.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the intermediate tridiagonal matrix T is scaled before computing the eigenvalues.

The default is to scale the tridiagonal matrix.

INIT (INPUT, OPTIONAL) logical(lgl) On entry, if INIT=true the initial intervals for the bisection steps for computing the eigenvalues of the intermediate tridiagonal matrix T are estimated from the eigenvalues of the intermediate tridiagonal matrix obtained from the Pal-Walker-Kahan algorithm.

The default is not to use the Pal-Walker-Kahan algorithm.

ABSTOL (INPUT, OPTIONAL) real(stnd) On entry, the absolute tolerance for the eigenvalues. An eigenvalue (or cluster) is considered to be located if it has been determined to lie in an interval whose width is ABSTOL or less.

If ABSTOL is less than or equal to zero, or is not specified, then ULP * | T | will be used, where | T | means the 1-norm of T (T is the intermediate tridiagonal form of MAT) and ULP is the machine precision (distance from 1 to the next larger floating point number).

Eigenvalues will be computed most accurately when ABSTOL is set to the square root of the underflow threshold, sqrt(LAMCH('S')), not zero.

LE (INPUT, OPTIONAL) integer(i4b) On entry, LE specifies the number of eigenvalues which must be computed by the subroutine. However, on output, NEIG may be different than LE if multiple eigenvalues at index LE make unique selection impossible.

If:

- SMALL=false, the subroutine computes the LE largest eigenvalues of MAT,
- SMALL=true, the subroutine computes the LE smallest eigenvalues of MAT.

Only one of the optional arguments LE and THETA must be specified, otherwise the subroutine will stop with an error message.

LE must be greater than 0 and less or equal to size(EIGVAL) .

The default is LE = size(EIGVAL), e.g. all the eigenvalues are computed.

THETA (INPUT, OPTIONAL) real(stnd) On entry,

- if SMALL=false, THETA specifies that the eigenvalues which are greater or equal to THETA must be computed. If none of the eigenvalues are greater or equal to THETA, NEIG is set to zero and EIGVAL(:) to a quiet NAN.
- if SMALL=true, THETA specifies that the eigenvalues which are less or equal to THETA must be computed. If none of the eigenvalues are smaller or equal to THETA, NEIG is set to zero and EIGVAL(:) to a quiet NAN.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of MAT. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of MAT. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.70 subroutine select_eigval_cmp3 (matp, neig, eigval, small, failure, upper, sort, vector, scaling, init, abstol, le, theta, d_e)

Purpose

SELECT_EIGVAL_CMP3 computes the largest or smallest eigenvalues of a n-by-n real symmetric matrix mat stored in packed form in a linear array MATP.

The matrix mat is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * mat * Q = T

,then the eigenvalues are computed by a bisection method.

Arguments

MATP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the upper or lower triangle of the symmetric matrix mat, packed column-wise in a linear array. The j-th column of mat is stored in the array MATP as follows:

- if UPPER = true, MATP(i + (j-1) * j/2) = mat(i,j) for 1<=i<=j;
- if UPPER = false, MATP(i + (j-1) * (2 * n-j)/2) = mat(i,j) for j<=i<=n.

On exit:

- If D_E is present : MATP is overwritten by the matrix Q as a product of elementary reflectors.
- If D_E is absent : MATP is destroyed.

The size of MATP must verify: size(MATP) = (n * (n+1)/2).

NEIG (OUTPUT) integer(i4b) On output, NEIG specifies the number of eigenvalues which have been computed. Note that NEIG may be greater than the optional argument LE, if multiple eigenvalues at index LE make unique selection impossible.

If none of the optional arguments LE and THETA are used, NEIG is set to n and all the eigenvalues of MAT are computed.

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, EIGVAL(1:NEIG) contains the first NEIG largest or smallest eigenvalues of MAT. The other values in EIGVAL (e.g. EIGVAL(NEIG+1:)) are flagged by a quiet NAN.

The size of EIGVAL must verify: size(EIGVAL) = n .

SMALL (INPUT) logical(lgl) On entry:

- SMALL = false : indicates that the m largest eigenvalues are desired.
- SMALL = true : indicates that the m smallest eigenvalues are desired.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit and the bisection algorithm converged for all the computed eigenvalues to the desired accuracy ;
- FAILURE = true : indicates that some or all of the eigenvalues failed to converge or were not computed. This is generally caused by unexpectedly inaccurate arithmetic.

UPPER (INPUT) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix mat is stored in the linear array MATP. If:

- UPPER = true : Upper triangle of mat is stored;
- UPPER = false: Lower triangle of mat is stored.

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. For other values of SORT nothing is done and EIGVAL(:NEIG) may not be sorted.

VECTOR (INPUT, OPTIONAL) logical(lgl) On entry, if VECTOR is set to true, a vectorized version of the bisection algorithm is used to find the eigenvalues of the intermediate tridiagonal form T of MAT.

The default is VECTOR=false.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the intermediate tridiagonal matrix T is scaled before computing the eigenvalues.

The default is to scale the tridiagonal matrix.

INIT (INPUT, OPTIONAL) logical(lgl) On entry, if INIT=true the initial intervals for the bisection steps for computing the eigenvalues of the intermediate tridiagonal matrix T are estimated from the eigenvalues of the intermediate tridiagonal matrix obtained from the Pal-Walker-Kahan algorithm.

The default is not to use the Pal-Walker-Kahan algorithm.

ABSTOL (INPUT, OPTIONAL) real(stnd) On entry, the absolute tolerance for the eigenvalues. An eigenvalue (or cluster) is considered to be located if it has been determined to lie in an interval whose width is ABSTOL or less.

If ABSTOL is less than or equal to zero, or is not specified, then ULP * | T | will be used, where | T | means the 1-norm of T (T is the intermediate tridiagonal form of mat) and ULP is the machine precision (distance from 1 to the next larger floating point number).

Eigenvalues will be computed most accurately when ABSTOL is set to the square root of the underflow threshold, sqrt(LAMCH('S')), not zero.

LE (INPUT, OPTIONAL) integer(i4b) On entry, LE specifies the number of eigenvalues which must be computed by the subroutine. However, on output, NEIG may be different than LE if multiple eigenvalues at index LE make unique selection impossible.

If:

- SMALL=false, the subroutine computes the LE largest eigenvalues of MAT,
- SMALL=true, the subroutine computes the LE smallest eigenvalues of MAT.

Only one of the optional arguments LE and THETA must be specified, otherwise the subroutine will stop with an error message.

LE must be greater than 0 and less or equal to size(EIGVAL) .

The default is LE = size(EIGVAL), e.g. all the eigenvalues are computed.

THETA (INPUT, OPTIONAL) real(stnd) On entry:

- if SMALL=false, THETA specifies that the eigenvalues which are greater or equal to THETA must be computed. If none of the eigenvalues are greater or equal to THETA, NEIG is set to zero and EIGVAL(:) to a quiet NAN.
- if SMALL=true, THETA specifies that the eigenvalues which are less or equal to THETA must be computed. If none of the eigenvalues are smaller or equal to THETA, NEIG is set to zero and EIGVAL(:) to a quiet NAN.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of mat. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of mat. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.71 subroutine select_eigval_cmp3 (matp, neig, eigval, small, failure, sort, vector, scaling, init, abstol, le, theta, d_e)

Purpose

SELECT_EIGVAL_CMP3 computes the largest or smallest eigenvalues of a n-by-n real symmetric matrix mat stored in packed form in a linear array MATP.

The matrix mat is first transformed to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * mat * Q = T

,then the eigenvalues are computed by a bisection method.

Arguments

MATP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the upper triangle of the symmetric matrix mat, packed column-wise in a linear array. The j-th column of mat is stored in the array MATP as follows:

MATP(i + (j-1) * j/2) = mat(i,j) for 1<=i<=j;

On exit:

- If D_E is present : MATP is overwritten by the matrix Q as a product of elementary reflectors.
- If D_E is absent : MATP is destroyed.

The size of MATP must verify: size(MATP) = (n * (n+1)/2).

NEIG (OUTPUT) integer(i4b) On output, NEIG specifies the number of eigenvalues which have been computed. Note that NEIG may be greater than the optional argument LE, if multiple eigenvalues at index LE make unique selection impossible.

If none of the optional arguments LE and THETA are used, NEIG is set to n and all the eigenvalues of MAT are computed.

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, EIGVAL(1:NEIG) contains the first NEIG largest or smallest eigenvalues of MAT. The other values in EIGVAL (e.g. EIGVAL(NEIG+1:)) are flagged by a quiet NAN.

The size of EIGVAL must verify: size(EIGVAL) = n .

SMALL (INPUT) logical(lgl) On entry:

- SMALL = false : indicates that the m largest eigenvalues are desired.
- SMALL = true : indicates that the m smallest eigenvalues are desired.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit and the bisection algorithm converged for all the computed eigenvalues to the desired accuracy ;
- FAILURE = true : indicates that some or all of the eigenvalues failed to converge or were not computed. This is generally caused by unexpectedly inaccurate arithmetic.

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. For other values of SORT nothing is done and EIGVAL(:NEIG) may not be sorted.

VECTOR (INPUT, OPTIONAL) logical(lgl) On entry, if VECTOR is set to true, a vectorized version of the bisection algorithm is used to find the eigenvalues of the intermediate tridiagonal form T of MAT.

The default is VECTOR=false.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the intermediate tridiagonal matrix T is scaled before computing the eigenvalues.

The default is to scale the tridiagonal matrix.

INIT (INPUT, OPTIONAL) logical(lgl) On entry, if INIT=true the initial intervals for the bisection steps for computing the eigenvalues of the intermediate tridiagonal matrix T are estimated from the eigenvalues of the intermediate tridiagonal matrix obtained from the Pal-Walker-Kahan algorithm.

The default is not to use the Pal-Walker-Kahan algorithm.

ABSTOL (INPUT, OPTIONAL) real(stnd) On entry, the absolute tolerance for the eigenvalues. An eigenvalue (or cluster) is considered to be located if it has been determined to lie in an interval whose width is ABSTOL or less.

If ABSTOL is less than or equal to zero, or is not specified, then ULP * | T | will be used, where | T | means the 1-norm of T (T is the intermediate tridiagonal form of MAT) and ULP is the machine precision (distance from 1 to the next larger floating point number).

Eigenvalues will be computed most accurately when ABSTOL is set to the square root of the underflow threshold, e.g., sqrt(LAMCH('S')), not zero.

LE (INPUT, OPTIONAL) integer(i4b) On entry, LE specifies the number of eigenvalues which must be computed by the subroutine. However, on output, NEIG may be different than LE if multiple eigenvalues at index LE make unique selection impossible.

If:

- SMALL=false, the subroutine computes the LE largest eigenvalues of MAT,
- SMALL=true, the subroutine computes the LE smallest eigenvalues of MAT.

Only one of the optional arguments LE and THETA must be specified, otherwise the subroutine will stop with an error message.

LE must be greater than 0 and less or equal to size(EIGVAL) .

The default is LE = size(EIGVAL), e.g., all the eigenvalues are computed.

THETA (INPUT, OPTIONAL) real(stnd) On entry:

- if SMALL=false, THETA specifies that the eigenvalues which are greater or equal to THETA must be computed. If none of the eigenvalues are greater or equal to THETA, NEIG is set to zero and EIGVAL(:) to a quiet NAN.
- if SMALL=true, THETA specifies that the eigenvalues which are less or equal to THETA must be computed. If none of the eigenvalues are smaller or equal to THETA, NEIG is set to zero and EIGVAL(:) to a quiet NAN.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

D_E (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit, the first column of D_E contains the n diagonal elements of the intermediate tridiagonal form T of mat. The n-1 first elements of the second column of D_E contains the n-1 subdiagonal elements of the intermediate tridiagonal form T of mat. D_E(n,2) is arbitrary.

The shape of D_E must verify: size(D_E, 1) = n and size(D_E, 2) = 2

Further Details

This driver subroutine is adapted from the routine DSYEV in LAPACK.

6.4.72 subroutine reig_cmp (mat, eigval, eigvec, failure, niter, nover, ortho, extd_samp, rng_alg, maxiter)

Purpose

REIG_CMP computes approximations of the neig largest eigenvalues (in absolute magnitude) and associated eigenvectors of a full n-by-n real symmetric matrix MAT using randomized power, subspace or block Krylov iterations.

neig is the target rank of the partial EigenValue Decomposition (EVD), which is sought, and is equal to the size of the output real vector argument EIGVAL, i.e., neig = size(EIGVAL).

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the n-by-n symmetric matrix MAT.

MAT is not modified by the routine.

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, EIGVAL(:) contains the first top neig eigenvalues of MAT. The eigenvalues are given in decreasing order of absolute magnitude.

The size of EIGVAL must verify:

- size(EIGVAL) = neig <= size(MAT, 1) = size(MAT, 2) = n.

EIGVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed neig top eigenvectors. The eigenvector associated with the eigenvalue EIGVAL(j) is stored in the j-th column of EIGVEC.

The shape of EIGVEC must verify:

- size(EIGVEC, 1) = size(MAT, 1) = size(MAT, 2) = n,
- size(EIGVEC, 2) = size(EIGVEC) = neig .

FAILURE (OUTPUT, OPTIONAL) logical(lgl) On exit, if the optional logical argument FAILURE is present, a test of the accuracy of the computed partial EVD is performed and in that case:

- FAILURE = false : indicates successful exit;
- FAILURE = true : indicates that some of the computed eigenvalues and eigenvectors of MAT failed to converge in NITER iterations.

If FAILURE = true on exit, results are still useful, but some of the approximated eigen couplets have a poor accuracy.

NITER (INPUT, OPTIONAL) integer(i4b) The number of randomized power, subspace or block Krylov iterations performed in the subroutine for computing the top neig eigen triplets. NITER must be positive or null.

By default, 10 randomized power, subspace or block Krylov iterations are performed.

NOVER (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the randomized power subspace or block Krylov iterations for computing the top neig eigen triplets.

NOVER must be positive or null and verifies the relationship:

- NOVER + size(EIGVAL) <= size(MAT, 1) = size(MAT, 2) = n

and is adjusted if necessary to verify this relationship in all cases.

See Further Details and the cited references for the meaning and usefulness of the oversampling size in the randomized power, subspace or block Krylov iterations.

By default, the oversampling size is set to 10.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, orthonormalization is carried out between each step of the power or block Krylov iterations to avoid loss of accuracy due to rounding errors. This means that subspace iterations are used instead of power iterations;
- ORTHO=false, orthonormalization is not performed.

The default is to use orthonormalization, e.g., ORTHO=true.

EXTD_SAMP (INPUT, OPTIONAL) logical(lgl) The optional argument EXTD_SAMP determines if extended sampling (e.g., block Krylov iterations) is used or not for computing the top neig eigen triplets.

On entry, if:

- EXTD_SAMP=true, block Krylov iterations are used;
- EXTD_SAMP=false, power or subspace iterations are used.

The default is to use power or subspace iterations, e.g., EXTD_SAMP=false.

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian test matrix in the randomized EVD algorithm.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to REIG_CMP.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm or in the QR phase of the EVD algorithm, which are used in the last phase of the randomized algorithm.

See description of suboutines SVD_CMP and EIG_CMP for further details about this optional argument.

Further Details

For a good introduction to randomized linear algebra, see the references (1) and (2).

The randomized subspace iteration was proposed in (3; see Algorithm 4.4) to compute an orthonormal matrix whose range approximates the range of MAT. An approximate partial EVD decomposition can then be computed using the aforementioned orthonormal matrix, see Algorithm 5.3 in (3).

The randomized block Krylov iterations for computing an approximate partial EVD was proposed in (4; see Algorithm 2). See also the reference (1).

For further details on randomized linear algebra, computing a partial EVD decomposition using randomized power, subspace or block Krylov iterations, see:

(1) **Martinsson, P.G., 2019:** Randomized methods for matrix computations. arXiv.1607.01649

(2) **Erichson, N.B., Voronin, S., Brunton, S.L., and Kutz, J.N., 2019:** Randomized matrix decompositions using R. arXiv.1608.02148

(3) **Halko, N., Martinsson, P.G., and Tropp, J.A., 2011:** Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions. SIAM Rev., 53, 217-288.

(4) **Musco, C., and Musco, C., 2015:** Randomized block krylov methods for stronger and faster approximate singular value decomposition. In Proceedings of the 28th International Conference on Neural Information Processing Systems, NIPS 15, pages 1396-1404, Cambridge, MA, USA, 2015. MIT Press.

(5) **Li, H.,Linderman, G.C., Szlam, A., Stanton, K.P., Kluger, Y., and Tygert, M., 2017:** Algorithm 971: An implementation of a randomized algorithm for principal component analysis. ACM Trans. Math. Softw. 43, 3, Article 28 (January 2017).

6.4.73 `function maxdiag_tinv_qr ( d, e, lambda )`

Purpose

This function computes the index of the element of maximum absolute value in the diagonal entries of (T - LAMBDA * I)**(-1) where T is a symmetric tridiagonal matrix, I is the identity matrix and LAMBDA is a scalar.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the tridiagonal matrix.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the tridiagonal matrix.

The size of E must verify: size(E) = size(D) - 1.

LAMBDA (INPUT) real(stnd) On entry, the eigenvalue or shift used in the QR factorization.

Further Details

The diagonal entries of (T - LAMBDA * I)**(-1) are computed by means of the QR factorization of (T - LAMBDA * I). For the latter computation, the semiseparable structure of (T - LAMBDA * I)**(-1) is used, see the reference (1). Moreover, it is assumed that T is unreduced, but no check is done in the subroutine to verify this assumption.

This subroutine is adapted from the pseudo-code trace_Tinv given in the reference (1).

For further details, see:

(1) **Bini, D.A., Gemignani, L., and Tisseur, F., 2005:** The Ehrlich-Aberth method for the nonsymmetric tridiagonal eigenvalue problem. SIAM J. Matrix Anal. Appl., 27, 153-175.

(2) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, 1013-1034.

(3) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

6.4.74 `function maxdiag_tinv_ldu ( d, e, lambda )`

Purpose

This function computes the index of the element of maximum absolute value in the diagonal entries of (T - LAMBDA * I)**(-1) where T is a symmetric tridiagonal matrix, I is the identity matrix and LAMBDA is a scalar.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the tridiagonal matrix.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the tridiagonal matrix.

The size of E must verify: size(E) = size(D) - 1.

LAMBDA (INPUT) real(stnd) On entry, the eigenvalue or shift used.

Further Details

The diagonal entries of (T - LAMBDA * I)**(-1) are computed by means of two triangular factorizations of (T - LAMBDA * I) of the forms L(+)D(+)U(+) and U(-)D(-)L(-) where L(+) and L(-) are unit lower bidiagonal, U(+) and U(-) are unit upper bidiagonal, and D(+) and D(-) are diagonal.

It is assumed that T is unreduced, but no check is done in the subroutine to verify this assumption.

This subroutine is adapted from the references (1) and (2).

For further details, on Fernando's method for computing eigenvectors of tridiagonal matrices, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(2) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

6.4.75 subroutine trid_qr_cmp (d, e, lambda, cs, sn, diag, sup1, sup2, maxdiag_tinv)

Purpose

TRID_QR_CMP factorizes the symmetric matrix T - LAMBDA * I, where T is an n by n symmetric tridiagonal matrix, I is the identity matrix and LAMBDA is a scalar, as

T - LAMBDA * I = Q * R

where Q is an orthogonal matrix represented as the product of n-1 Givens rotations and R is an upper triangular matrix with at most two non-zero super-diagonal elements per column.

The parameter LAMBDA is included in the routine so that TRID_QR_CMP may be used to obtain eigenvectors of T by inverse iteration.

The subroutine also computes the index of the entry of maximum absolute value in the diagonal of (T - LAMBDA * I)**(-1), which provides a good initial approximation to start the inverse iteration process for computing the eigenvector assoociated with the eigenvalue LAMBDA, see the references (1), (2) and (3) for further details.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the tridiagonal matrix.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the tridiagonal matrix.

The size of E must verify: size(E) = size(D) - 1.

LAMBDA (INPUT) real(stnd) On entry, the eigenvalue or shift used in the QR factorization.

CS (OUTPUT) real(stnd), dimension(:) On exit, the vector of the cosines coefficients of the chain of n-1 Givens rotations for the QR factorization of T - LAMBDA * I.

The size of CS must verify: size(CS) = size(E) = size(D) - 1.

SN (OUTPUT) real(stnd), dimension(:) On exit, the vector of the sines coefficients of the chain of n-1 Givens rotations for the QR factorization of T - LAMBDA * I.

The size of SN must verify: size(SN) = size(E) = size(D) - 1.

DIAG (OUTPUT) real(stnd), dimension(:) On exit, DIAG(:) contains the n diagonal elements of the upper triangular matrix R of the QR factorization of T - LAMBDA * I.

The size of DIAG must verify: size(DIAG) = size(D) = n .

SUP1 (OUTPUT) real(stnd), dimension(:) On exit, SUP1(:n-1) contains the n-1 superdiagonal elements of the upper triangular matrix R of the QR factorization of T - LAMBDA * I, SUP1(n) is arbitrary.

The size of SUP1 must verify: size(SUP1) = size(D) = n .

SUP2 (OUTPUT) real(stnd), dimension(:) On exit, SUP2(:n-2) contains the n-2 second superdiagonal elements of the upper triangular matrix R of the QR factorization of T - LAMBDA * I, SUP2(n-1:n) is arbitrary.

The size of SUP2 must verify: size(SUP2) = size(D) = n .

MAXDIAG_TINV (OUPTPUT) integer(i4b) On exit, MAXDIAG_TINV is the index of the entry of maximum modulus in the main diagonal of (T - LAMBDA * I)**(-1).

Further Details

The QR factorization of (T - LAMBDA * I) is obtained by means of n-1 unitary Givens rotations.

The diagonal entries of (T - LAMBDA * I)**(-1) are computed by means of this QR factorization of (T - LAMBDA * I). For the latter computation, the semiseparable structure of (T - LAMBDA * I)**(-1) is used, see the reference (1). Moreover, it is assumed that T is unreduced for computing the index of the entry of maximum absolute value in the diagonal of (T - LAMBDA * I)**(-1), but no check is done in the subroutine to verify this assumption.

For further details, see:

(1) **Bini, D.A., Gemignani, L., and Tisseur, F., 2005:** The Ehrlich-Aberth method for the nonsymmetric tridiagonal eigenvalue problem. SIAM J. Matrix Anal. Appl., 27, 153-175.

(2) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(3) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

6.4.76 `subroutine trid_qr_solve ( cs, sn, diag, sup1, sup2, y )`

Purpose

TRID_QR_SOLVE may be used to solve for x(:) the system of equations

x(:) * (T - LAMBDA * I) = scale * y(:)

, where T is an n-by-n symmetric tridiagonal matrix, I is the n-by-n identity matrix, LAMBDA and scale are scalars, following the factorization of (T - LAMBDA * I) by TRID_QR_CMP or GK_QR_CMP, as

T - LAMBDA * I = Q * R

where Q is an orthogonal matrix represented as the product of n-1 Givens rotations and R is an upper triangular matrix with at most two non-zero super-diagonal elements per column.

The matrix (T - LAMBDA * I) is assumed to be ill-conditioned, and frequent rescalings are carried out in order to avoid overflow. However, no test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine. The scalar, scale, is not output by this routine since this routine being intended for use in applications such as inverse iteration.

Arguments

CS (INPUT) real(stnd), dimension(:) On entry, the vector of the cosines coefficients of the chain of n-1 Givens rotations for the QR factorization of T - LAMBDA * I as computed by TRID_QR_CMP or GK_QR_CMP.

The size of CS must verify: size(CS) = size(Y) - 1.

SN (INPUT) real(stnd), dimension(:) On entry, the vector of the sines coefficients of the chain of n-1 Givens rotations for the QR factorization of GK - LAMBDA * I as computed by TRID_QR_CMP or GK_QR_CMP.

The size of SN must verify: size(SN) = size(Y) - 1.

DIAG (INPUT) real(stnd), dimension(:) On entry, DIAG(:) contains the n diagonal elements of the upper triangular matrix R of the QR factorization of T - LAMBDA * I.

The size of DIAG must verify: size(DIAG) = size(Y) = n .

SUP1 (INPUT) real(stnd), dimension(:) On entry, SUP1(:n-1) contains the n-1 superdiagonal elements of the upper triangular matrix R of the QR factorization of T - LAMBDA * I, SUP1(n) is arbitrary.

The size of SUP1 must verify: size(SUP1) = size(Y) = n .

SUP2 (INPUT) real(stnd), dimension(:) On entry, SUP2(:n-2) contains the n-2 second superdiagonal elements of the upper triangular matrix R of the QR factorization of T - LAMBDA * I, SUP2(n-1:n) is arbitrary.

The size of SUP2 must verify: size(SUP2) = size(Y) = n .

Y (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the right hand side vector y. On exit, Y is overwritten the solution vector x.

The size of Y must verify: size(Y) = n .

Further Details

For further details, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(2) **Bini, D.A., Gemignani, L., and Tisseur, F., 2005:** The Ehrlich-Aberth method for the nonsymmetric tridiagonal eigenvalue problem. SIAM J. Matrix Anal. Appl., 27, 153-175.

6.4.77 `subroutine trid_cmp ( d, e, eigval, sub, diag, sup1, sup2, perm, tol )`

Purpose

TRID_CMP factorizes symmetric matrices of the form (T - EIGVAL(j) * I), where T is an n-by-n symmetric tridiagonal matrix and EIGVAL(j) is a scalar, as

T - EIGVAL(j) * I = P(j) * L(j) * U(j), for j=1, SIZE(EIGVAL)

where P(j) is a permutation matrix, L(j) is a unit lower tridiagonal matrix with at most one non-zero sub-diagonal elements per column and U(j) is an upper triangular matrix with at most two non-zero super-diagonal elements per column.

The factorizations, for j=1, SIZE(EIGVAL), are obtained by Gaussian elimination with partial pivoting and implicit row scaling.

The parameters EIGVAL are included in the routine so that TRID_CMP may be used to obtain eigenvectors of T by inverse iteration.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the symmetric tridiagonal matrix T.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the symmetric tridiagonal matrix T and E(n) is arbitrary .

The size of E must verify: size(E) = size(D) = n .

EIGVAL (INPUT) real(stnd), dimension(:) On entry, selected eigenvalues of the symmetric tridiagonal matrix.

The size of EIGVAL must verify: size(EIGVAL) <= size(D) = n .

SUB (OUTPUT) real(stnd), dimension(:,:) On exit, SUB(j,:n-1) contains the n-1 subdiagonal elements of the lower triangular matrix L(j) of the factorization of T - EIGVAL(j) * I, for j=1, SIZE(EIGVAL). SUB(:,n) is arbitrary .

The shape of SUB must verify:

- size(SUB, 1) = size(EIGVAL) ;
- size(SUB, 2) = size(D) = n .

DIAG (OUTPUT) real(stnd), dimension(:,:) On exit, DIAG(j,:) contains the n diagonal elements of the upper triangular matrix U(j) of the factorization of T - EIGVAL(j) * I, for j=1, SIZE(EIGVAL).

The shape of DIAG must verify:

- size(DIAG, 1) = size(EIGVAL) ;
- size(DIAG, 2) = size(D) = n .

SUP1 (OUTPUT) real(stnd), dimension(:,:) On exit, SUP1(j,:n-1) contains the n-1 superdiagonal elements of the upper triangular matrix U(j) of the factorization of T - EIGVAL(j) * I, for j=1, SIZE(EIGVAL). SUP1(:,n) is arbitrary .

The shape of SUP1 must verify:

- size(SUP1, 1) = size(EIGVAL) ;
- size(SUP1, 2) = size(D) = n .

SUP2 (OUTPUT) real(stnd), dimension(:,:) On exit, SUP2(j,:n-2) contains the n-2 second superdiagonal elements of the upper triangular matrix U(j) of the factorization of T - EIGVAL(j) * I, for j=1, SIZE(EIGVAL). SUP2(:,n-1:n) is arbitrary .

The shape of SUP2 must verify:

- size(SUP2, 1) = size(EIGVAL) ;
- size(SUP2, 2) = size(D) = n .

PERM (OUTPUT) logical(lgl), dimension(:,:) On exit, PERM(j,:n-1) contains details of the permutation matrix P(j). If an interchange occurred at the kth step of the elimination in the factorization of (T - EIGVAL(j) * I), then PERM(j,k) = true, otherwise PERM(j,k) = false. The element PERM(j,n) is set to true if there is an integer l such that

abs(u(j)(l,l)).le. norm((T - EIGVAL(j) * I)(l)) * TOL,

where norm(A(l)) denotes the sum of the absolute values of the lth row of the matrix A. If no such l exists then PERM(j,n) is returned as false. If PERM(j,n) is returned as true, then a diagonal element of U(j) is small, indicating that (T - EIGVAL(j) * I) is singular or nearly singular.

The shape of PERM must verify:

- size(PERM, 1) = size(EIGVAL) ;
- size(PERM, 2) = size(D) = n .

TOL (INPUT, OPTIONAL) real(stnd) On entry, a relative tolerance used to indicate whether or not the matrices (T - EIGVAL(j) * I) are nearly singular. TOL should normally be choose as approximately the largest relative error in the elements of T. For example, if the elements of T are correct to about 4 significant figures, then TOL should be set to about 5 * 10**(-4).

If TOL is supplied as less than eps, where eps is the relative machine precision, then the value eps is used in place of TOL.

Further Details

This subroutine is adapted from the routine DLAGTF in LAPACK.

6.4.78 subroutine trid_cmp (d, e, eigval, sub, diag, sup1, sup2, perm, tol)

Purpose

TRID_CMP factorizes the symmetric matrix (T - EIGVAL * I), where T is an n-by-n symmetric tridiagonal matrix and EIGVAL is a scalar, as

T - EIGVAL * I = P * L * U

where P is a permutation matrix, L is a unit lower tridiagonal matrix with at most one non-zero subdiagonal elements per column and U is an upper triangular matrix with at most two non-zero superdiagonal elements per column.

The factorization is obtained by Gaussian elimination with partial pivoting and implicit row scaling.

The parameter EIGVAL is included in the routine so that TRID_CMP may be used to obtain eigenvectors of T by inverse iteration.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the symmetric tridiagonal matrix T.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the symmetric tridiagonal matrix T and E(n) is arbitrary.

The size of E must verify: size(E) = size(D) = n .

EIGVAL (INPUT) real(stnd) On entry, an eigenvalue of the symmetric tridiagonal matrix.

SUB (OUTPUT) real(stnd), dimension(:) On exit, SUB(:n-1) contains the n-1 subdiagonal elements of the lower triangular matrix L of the factorization of T - EIGVAL * I, SUB(n) is arbitrary .

The size of SUB must verify: size(SUB) = size(D) = n .

DIAG (OUTPUT) real(stnd), dimension(:) On exit, DIAG(:) contains the n diagonal elements of the upper triangular matrix U of the factorization of T - EIGVAL * I.

The size of DIAG must verify: size(DIAG) = size(D) = n .

SUP1 (OUTPUT) real(stnd), dimension(:) On exit, SUP1(:n-1) contains the n-1 superdiagonal elements of the upper triangular matrix U of the factorization of T - EIGVAL * I, SUP1(n) is arbitrary.

The size of SUP1 must verify: size(SUP1) = size(D) = n .

SUP2 (OUTPUT) real(stnd), dimension(:) On exit, SUP2(:n-2) contains the n-2 second superdiagonal elements of the upper triangular matrix U of the factorization of T - EIGVAL * I, SUP2(n-1:n) is arbitrary .

The size of SUP2 must verify: size(SUP2) = size(D) = n .

PERM (OUTPUT) logical(lgl), dimension(:) On exit, PERM(:n-1) contains details of the permutation matrix P(j). If an interchange occurred at the kth step of the elimination in the factorization of (T - EIGVAL(j) * I), then PERM(k) = true, otherwise PERM(k) = false. The element PERM(n) is set to true if there is an integer l such that

abs(u(l,l)).le. norm((T - EIGVAL * I)(l)) * TOL,

where norm(A(l)) denotes the sum of the absolute values of the lth row of the matrix A. If no such l exists then PERM(n) is returned as false. If PERM(n) is returned as true, then a diagonal element of U is small, indicating that (T - EIGVAL * I) is singular or nearly singular.

The size of PERM must verify: size(PERM) = size(D) = n .

TOL (INPUT, OPTIONAL) real(stnd) On entry, a relative tolerance used to indicate whether or not the matrix (T - EIGVAL * I) is nearly singular. TOL should normally be choose as approximately the largest relative error in the elements of T. For example, if the elements of T are correct to about 4 significant figures, then TOL should be set to about 5 * 10**(-4).

If TOL is supplied as less than eps, where eps is the relative machine precision, then the value eps is used in place of TOL.

Further Details

This subroutine is adapted from the routine DLAGTF in LAPACK.

6.4.79 subroutine trid_cmp2 (d, e, eigval, sub, diag, sup1, sup2, perm)

Purpose

TRID_CMP2 factorizes symmetric matrices of the form (T - EIGVAL(j) * I), where T is an n-by-n symmetric tridiagonal matrix and EIGVAL(j) is a scalar, as

T - EIGVAL(j) * I = P(j) * L(j) * U(j), for j=1, SIZE(EIGVAL)

where P(j) is a permutation matrix, L(j) is a unit lower tridiagonal matrix with at most one non-zero sub-diagonal elements per column and U(j) is an upper triangular matrix with at most two non-zero super-diagonal elements per column.

The factorizations, for j=1, SIZE(EIGVAL), are obtained by Gaussian elimination with partial pivoting and row interchanges.

The parameters EIGVAL are included in the routine so that TRID_CMP2 may be used to obtain eigenvectors of T by inverse iteration.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the symmetric tridiagonal matrix T.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the symmetric tridiagonal matrix T and E(n) is arbitrary .

The size of E must verify: size(E) = size(D) = n .

EIGVAL (INPUT) real(stnd), dimension(:) On entry, selected eigenvalues of the symmetric tridiagonal matrix.

The size of EIGVAL must verify: size(EIGVAL) <= size(D) = n .

SUB (OUTPUT) real(stnd), dimension(:,:) On exit, SUB(j,:n-1) contains the n-1 subdiagonal elements of the lower triangular matrix L(j) of the factorization of T - EIGVAL(j) * I, for j=1, SIZE(EIGVAL). SUB(:,n) is arbitrary .

The shape of SUB must verify:

- size(SUB, 1) = size(EIGVAL) ;
- size(SUB, 2) = size(D) = n .

DIAG (OUTPUT) real(stnd), dimension(:,:) On exit, DIAG(j,:) contains the n diagonal elements of the upper triangular matrix U(j) of the factorization of T - EIGVAL(j) * I, for j=1, SIZE(EIGVAL).

The shape of DIAG must verify:

- size(DIAG, 1) = size(EIGVAL) ;
- size(DIAG, 2) = size(D) = n .

SUP1 (OUTPUT) real(stnd), dimension(:,:) On exit, SUP1(j,:n-1) contains the n-1 superdiagonal elements of the upper triangular matrix U(j) of the factorization of T - EIGVAL(j) * I, for j=1, SIZE(EIGVAL). SUP1(:,n) is arbitrary .

The shape of SUP1 must verify:

- size(SUP1, 1) = size(EIGVAL) ;
- size(SUP1, 2) = size(D) = n .

SUP2 (OUTPUT) real(stnd), dimension(:,:) On exit, SUP2(j,:n-2) contains the n-2 second superdiagonal elements of the upper triangular matrix U(j) of the factorization of T - EIGVAL(j) * I, for j=1, SIZE(EIGVAL). SUP2(:,n-1:n) is arbitrary .

The shape of SUP2 must verify:

- size(SUP2, 1) = size(EIGVAL) ;
- size(SUP2, 2) = size(D) = n .

PERM (OUTPUT) logical(lgl), dimension(:,:) On exit, PERM(j,:n-1) contains details of the permutation matrix P(j). If an interchange occurred at the kth step of the elimination in the factorization of (T - EIGVAL(j) * I), then PERM(j,k) = true, otherwise PERM(j,k) = false. PERM(:,n) is arbitrary .

The shape of PERM must verify:

- size(PERM, 1) = size(EIGVAL) ;
- size(PERM, 2) = size(D) = n .

Further Details

TRID_CMP2 is a simplified version of TRID_CMP. This subroutine is adapted from the routine DGTTRF in LAPACK.

6.4.80 subroutine trid_cmp2 (d, e, eigval, sub, diag, sup1, sup2, perm)

Purpose

TRID_CMP2 factorizes the symmetric matrix (T - EIGVAL * I), where T is an n by n symmetric tridiagonal matrix and EIGVAL is a scalar, as

T - EIGVAL * I = P * L * U

where P is a permutation matrix, L is a unit lower tridiagonal matrix with at most one non-zero subdiagonal elements per column and U is an upper triangular matrix with at most two non-zero superdiagonal elements per column.

The factorization is obtained by Gaussian elimination with partial pivoting and row interchanges.

The parameter EIGVAL is included in the routine so that TRID_CMP2 may be used to obtain eigenvectors of T by inverse iteration.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the symmetric tridiagonal matrix T.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the symmetric tridiagonal matrix T and E(n) is arbitrary .

The size of E must verify: size(E) = size(D) = n .

EIGVAL (INPUT) real(stnd) On entry, an eigenvalue of the symmetric tridiagonal matrix.

SUB (OUTPUT) real(stnd), dimension(:) On exit, SUB(:n-1) contains the n-1 subdiagonal elements of the lower triangular matrix L of the factorization of T - EIGVAL * I, SUB(n) is arbitrary .

The size of SUB must verify: size(SUB) = size(D) = n .

DIAG (OUTPUT) real(stnd), dimension(:) On exit, DIAG(:) contains the n diagonal elements of the upper triangular matrix U of the factorization of T - EIGVAL * I.

The size of DIAG must verify: size(DIAG) = size(D) = n .

SUP1 (OUTPUT) real(stnd), dimension(:) On exit, SUP1(:n-1) contains the n-1 superdiagonal elements of the upper triangular matrix U of the factorization of T - EIGVAL * I, SUP1(n) is arbitrary .

The size of SUP1 must verify: size(SUP1) = size(D) = n .

SUP2 (OUTPUT) real(stnd), dimension(:) On exit, SUP2(:n-2) contains the n-2 second superdiagonal elements of the upper triangular matrix U of the factorization of T - EIGVAL * I, SUP2(n-1:n) is arbitrary .

The size of SUP2 must verify: size(SUP2) = size(D) = n .

PERM (OUTPUT) logical(lgl), dimension(:) On exit, PERM(:n-1) contains details of the permutation matrix P(j). If an interchange occurred at the kth step of the elimination in the factorization of (T - EIGVAL(j) * I), then PERM(k) = true, otherwise PERM(k) = false. PERM(n) is arbitrary .

The size of PERM must verify: size(PERM) = size(D) = n .

Further Details

TRID_CMP2 is a simplified version of TRID_CMP. This subroutine is adapted from the routine DGTTRF in LAPACK.

6.4.81 subroutine trid_solve (sub, diag, sup1, sup2, perm, y)

Purpose

TRID_SOLVE may be used to solve systems of equations of the form

x(j,:) * (T - EIGVAL(j) * I) = scale(j) * y(j,:), for j=1, SIZE(EIGVAL)

, where T is an n by n symmetric tridiagonal matrix, EIGVAL(j) and scale(j) are scalars, for x(j,:) for j=1, SIZE(EIGVAL), following the factorization of (T - EIGVAL(j) * I) by TRID_CMP or TRID_CMP2 as

T - EIGVAL(j) * I = P(j) * L(j) * U(j), for j=1, SIZE(EIGVAL)

where P(j) is a permutation matrix, L(j) is a unit lower tridiagonal matrix with at most one non-zero sub-diagonal elements per column and U(j) is an upper triangular matrix with at most two non-zero super-diagonal elements per column.

The matrices (T - EIGVAL(j) * I) are assumed to be ill-conditioned, and frequent rescalings are carried out in order to avoid overflow. However, no test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine. The scalars, scale(j) for j=1, SIZE(EIGVAL), are not output by this routine since this routine being intended for use in applications such as inverse iteration.

Arguments

SUB (INPUT) real(stnd), dimension(:,:) On entry, SUB(j,:n-1) contains the n-1 subdiagonal elements of the lower triangular matrix L(j) of the factorization of T - EIGVAL(j) * I, for j=1, SIZE(EIGVAL). SUB(:,n) is arbitrary .

The shape of SUB must verify:

- size(SUB, 1) = size(Y, 1) = size(EIGVAL) ;
- size(SUB, 2) = size(Y, 2) = n .

DIAG (INPUT) real(stnd), dimension(:,:) On entry, DIAG(j,:) contains the n diagonal elements of the upper triangular matrix U(j) of the factorization of T - EIGVAL(j) * I, for j=1, SIZE(EIGVAL).

The shape of DIAG must verify:

- size(DIAG, 1) = size(Y, 1) = size(EIGVAL) ;
- size(DIAG, 2) = size(Y, 2) = n .

SUP1 (INPUT) real(stnd), dimension(:,:) On entry, SUP1(j,:n-1) contains the n-1 superdiagonal elements of the upper triangular matrix U(j) of the factorization of T - EIGVAL(j) * I, for j=1, SIZE(EIGVAL). SUP1(:,n) is arbitrary.

The shape of SUP1 must verify:

- size(SUP1, 1) = size(Y, 1) = size(EIGVAL) ;
- size(SUP1, 2) = size(Y, 2) = n .

SUP2 (INPUT) real(stnd), dimension(:,:) On entry, SUP2(j,:n-2) contains the n-2 second superdiagonal elements of the upper triangular matrix U(j) of the factorization of T - EIGVAL(j) * I, for j=1, SIZE(EIGVAL). SUP2(:,n-1:n) is arbitrary.

The shape of SUP2 must verify:

- size(SUP2, 1) = size(Y, 1) = size(EIGVAL) ;
- size(SUP2, 2) = size(Y, 2) = n .

PERM (INPUT) logical(lgl), dimension(:,:) On entry, PERM(j,:n-1) contains details of the permutation matrix P(j). If an interchange occurred at the kth step of the elimination in the factorization of (T - EIGVAL(j) * I), then PERM(j,k) = true, otherwise PERM(j,k) = false. PERM(:,n) is arbitrary .

The shape of PERM must verify:

- size(PERM, 1) = size(Y, 1) = size(EIGVAL) ;
- size(PERM, 2) = size(Y, 2) = n .

Y (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the right hand side matrix y. On exit, Y is overwritten the solution matrix x.

The shape of Y must verify:

- size(Y, 1) = size(EIGVAL) ;
- size(Y, 2) = n .

Further Details

This subroutine is adapted from the routine DLAGTS in LAPACK.

6.4.82 subroutine trid_solve (sub, diag, sup1, sup2, perm, y)

Purpose

TRID_SOLVE may be used to solve the system of equations

x(:) * (T - EIGVAL * I) = scale * y(:)

, where T is an n by n symmetric tridiagonal matrix, EIGVAL and scale are scalars, for x(:), following the factorization of (T - EIGVAL * I) by TRID_CMP or TRID_CMP2 as

T - EIGVAL * I = P * L * U

where P is a permutation matrix, L is a unit lower tridiagonal matrix with at most one non-zero subdiagonal elements per column and U is an upper triangular matrix with at most two non-zero superdiagonal elements per column.

The matrix (T - EIGVAL * I) is assumed to be ill-conditioned, and frequent rescalings are carried out in order to avoid overflow. However, no test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine. The scalar, scale, is not output by this routine since this routine being intended for use in applications such as inverse iteration.

Arguments

SUB (INPUT) real(stnd), dimension(:) On entry, SUB(:n-1) contains the n-1 subdiagonal elements of the lower triangular matrix L of the factorization of T - EIGVAL * I, SUB(n) is arbitrary .

The size of SUB must verify: size(SUB) = size(Y) = n .

DIAG (INPUT) real(stnd), dimension(:) On entry, DIAG(:) contains the n diagonal elements of the upper triangular matrix U of the factorization of T - EIGVAL * I.

The shape of DIAG must verify: size(DIAG) = size(Y) = n .

SUP1 (INPUT) real(stnd), dimension(:) On entry, SUP1(:n-1) contains the n-1 superdiagonal elements of the upper triangular matrix U of the factorization of T - EIGVAL * I, SUP1(n) is arbitrary.

The shape of SUP1 must verify: size(SUP1) = size(Y) = n .

SUP2 (INPUT) real(stnd), dimension(:) On entry, SUP2(:n-2) contains the n-2 second superdiagonal elements of the upper triangular matrix U of the factorization of T - EIGVAL * I, SUP2(n-1:n) is arbitrary.

The shape of SUP2 must verify: size(SUP2) = size(Y) = n .

PERM (INPUT) logical(lgl), dimension(:) On entry, PERM(:n-1) contains details of the permutation matrix P. If an interchange occurred at the kth step of the elimination in the factorization of (T - EIGVAL * I), then PERM(k) = true, otherwise PERM(k) = false. PERM(n) is arbitrary .

The shape of PERM must verify: size(PERM) = size(Y) = n .

Y (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the right hand side vector y. On exit, Y is overwritten the solution vector x.

The shape of Y must verify: size(Y) = n .

Further Details

This subroutine is adapted from the routine DLAGTS in LAPACK.

6.4.83 subroutine trid_inviter (d, e, eigval, eigvec, failure, maxiter, scaling, initvec)

Purpose

TRID_INVITER computes an eigenvector of a real n-by-n symmetric tridiagonal matrix T corresponding to a specified eigenvalue, by combining Fernando's method for computing an eigenvector of a real n-by-n symmetric tridiagonal matrix and an inverse iteration algorithm.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the symmetric tridiagonal matrix T.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the symmetric tridiagonal matrix T and E(n) is arbitrary .

The size of E must verify: size(E) = size(D) = n .

EIGVAL (INPUT) real(stnd) On entry, an eigenvalue of the symmetric tridiagonal matrix.

EIGVEC (OUTPUT) real(stnd), dimension(:) On exit, the computed eigenvector.

The shape of EIGVEC must verify: size(EIGVEC) = size(D) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false indicates successful exit.
- FAILURE = true indicates that the eigenvector failed to converge in MAXITER iterations.

MAXITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed in the subroutine.

By default, 2 inverse iterations are performed.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the tridiagonal matrix T is scaled before computing the eigenvector.

The default is to scale the tridiagonal matrix.

INITVEC (INPUT, OPTIONAL) logical(lgl) On entry, if INITVEC=true a Fernando vector is used to start the inverse iteration process; if INITVEC=false a random uniform starting vector is used.

For unreduced tridiagonal matrices, the default is to use a Fernando starting vector. For reduced tridiagonal matrices, the default is to use a random uniform starting vector.

Further Details

TRID_INVITER uses Fernando's method for computing a first estimate of an eigenvector corresponding to an approximate eigenvalue of a real n-by-n symmetric tridiagonal matrix T (by default, only if the input tridiagonal matrix T is unreduced).

This approximate eigenvector is then refined (or computed if Fernando's method is not used) using an inverse iteration algorithm.

For further details, on Fernando's method for computing eigenvectors of tridiagonal matrices or inverse iteration, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(2) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

(3) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(4) **Bini, D.A., Gemignani, L., and Tisseur, F., 2005:** The Ehrlich-Aberth method for the nonsymmetric tridiagonal eigenvalue problem. SIAM J. Matrix Anal. Appl., 27, 153-175.

6.4.84 `subroutine trid_inviter ( d, e, eigval, eigvec, failure, maxiter, ortho, backward_sweep, scaling, initvec )`

Purpose

TRID_INVITER computes the eigenvectors of a real n-by-n symmetric tridiagonal matrix T corresponding to specified eigenvalues, by combining Fernando's method for computing (selected) eigenvectors of a real n-by-n symmetric tridiagonal matrix and an inverse iteration algorithm.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the symmetric tridiagonal matrix T.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the symmetric tridiagonal matrix T and E(n) is arbitrary .

The size of E must verify: size(E) = size(D) = n .

EIGVAL (INPUT) real(stnd), dimension(:) On entry, selected eigenvalues of the symmetric tridiagonal matrix. The eigenvalues must be given in decreasing order.

The size of EIGVAL must verify: size(EIGVAL) <= size(D) = n .

EIGVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed eigenvectors. The eigenvector associated with the eigenvalue EIGVAL(j) is stored in the j-th column of EIGVEC.

The shape of EIGVEC must verify:

- size(EIGVEC, 1) = size(D) = n ,
- size(EIGVEC, 2) = size(EIGVAL) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false indicates successful exit.
- FAILURE = true indicates that some eigenvectors failed to converge in MAXITER iterations.

MAXITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed in the subroutine. By default, 2 inverse iterations are performed for all the eigenvectors.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, all the eigenvectors are orthogonalized by the Modified Gram-Schmidt or QR algorithm;
- ORTHO=false, the eigenvectors are not orthogonalized by the Modified Gram-Schmidt or QR algorithm.

The default is to orthogonalize the eigenvectors only for the eigenvalues, which are not well-separated.

BACKWARD_SWEEP (INPUT, OPTIONAL) logical(lgl) On entry, if:

- BACKWARD_SWEEP=true and the eigenvectors are orthogonalized by the modified Gram-Schmidt algorithm, a backward sweep of the modified Gram-Schmidt algorithm is also performed;
- BACKWARD_SWEEP=false a backward sweep is not performed.

The default is not to perform a backward sweep of the modified Gram-Schmidt algorithm.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if:

- SCALING=true, the tridiagonal matrix T is scaled before computing the eigenvectors;
- SCALING=false, the tridiagonal matrix T is not scaled.

The default is to scale the tridiagonal matrix.

INITVEC (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITVEC=true, Fernando vectors are used to start the inverse iteration process;
- INITVEC=false, random uniform starting vectors are used.

For unreduced tridiagonal matrices, the default is to use Fernando starting vectors if the eigenvalues are well-separated and random uniform starting vectors otherwise. For reduced tridiagonal matrices, the default is to use random uniform starting vectors.

Further Details

TRID_INVITER uses Fernando's method for computing a first estimate of (selected) eigenvectors corresponding to (selected) approximate eigenvalues of a real n-by-n symmetric tridiagonal matrix T (by default, only for the eigenvalues which are well separated and if the input tridiagonal matrix T is unreduced).

These approximate eigenvectors are then refined (or computed if Fernando's method is not used) using an inverse iteration algorithm for all the eigenvalues at one step. The eigenvectors are then orthogonalized by the Modified Gram-Schmidt or QR algorithm if the eigenvalues are not well-separated.

TRID_INVITER may fail if some the eigenvalues specified in parameter EIGVAL are nearly identical.

For further details, on Fernando's method for computing eigenvectors of tridiagonal matrices or inverse iteration, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(2) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

(3) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(4) **Bini, D.A., Gemignani, L., and Tisseur, F., 2005:** The Ehrlich-Aberth method for the nonsymmetric tridiagonal eigenvalue problem. SIAM J. Matrix Anal. Appl., 27, 153-175.

6.4.85 subroutine trid_inviter (d, e, eigval, eigvec, failure, mat, maxiter, ortho, backward_sweep, scaling, initvec)

Purpose

TRID_INVITER computes the eigenvectors of a full real n-by-n symmetric matrix MAT corresponding to specified eigenvalues, by combining Fernando's method for computing (selected) eigenvectors of a real n-by-n symmetric tridiagonal matrix and inverse iteration followed by a back-transformation procedure.

It is required that the original symmetric matrix MAT has been reduced to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

with a call to SYMTRID_CMP with parameter STORE_Q set to true, before calling TRID_INVITER.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the symmetric tridiagonal form T of MAT.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the symmetric tridiagonal form T of MAT. E(n) is arbitrary .

The size of E must verify: size(E) = size(D) = n .

EIGVAL (INPUT) real(stnd), dimension(:) On entry, selected eigenvalues of the symmetric matrix MAT. The eigenvalues must be given in decreasing order.

The size of EIGVAL must verify: size(EIGVAL) <= size(D) = n .

EIGVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed eigenvectors. The eigenvector associated with the eigenvalue EIGVAL(j) is stored in the j-th column of EIGVEC.

The shape of EIGVEC must verify:

- size(EIGVEC, 1) = size(D) = n ;
- size(EIGVEC, 2) = size(EIGVAL) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false indicates successful exit.
- FAILURE = true indicates that some eigenvectors failed to converge in MAXITER iterations.

MAT (INPUT) real(stnd), dimension(:,:) On entry, the vectors and the scalars which define the elementary reflectors used to reduce the full real n-by-n symmetric matrix MAT to symmetric tridiagonal form T, as returned by SYMTRID_CMP with STORE_Q=true, in its argument MAT. MAT is not modified by the routine.

Back-transformation is used to find the selected eigenvectors of the original matrix MAT and these eigenvectors are stored in argument EIGVEC.

The shape of MAT must verify:

- size(MAT, 1) = size(D) = n ;
- size(MAT, 2) = size(D) = n .

MAXITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed in the subroutine. By default, 2 inverse iterations are performed for all the eigenvectors of the tridiagonal matrix T.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, all the eigenvectors are orthogonalized by the Modified Gram-Schmidt or QR algorithm;
- ORTHO=false, the eigenvectors are not orthogonalized by the Modified Gram-Schmidt or QR algorithm.

The default is to orthogonalize the eigenvectors only if the eigenvalues are not well-separated.

BACKWARD_SWEEP (INPUT, OPTIONAL) logical(lgl) On entry, if:

- BACKWARD_SWEEP=true and the eigenvectors are orthogonalized by the modified Gram-Schmidt algorithm, a backward sweep of the modified Gram-Schmidt algorithm is also performed;
- BACKWARD_SWEEP=false a backward sweep is not performed.

The default is not to perform a backward sweep of the modified Gram-Schmidt algorithm.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if:

- SCALING=true, the tridiagonal matrix T is scaled before computing the eigenvectors;
- SCALING=false, the tridiagonal matrix T is not scaled.

The default is to scale the tridiagonal matrix.

INITVEC (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITVEC=true, Fernando vectors are used to start the inverse iteration process;
- INITVEC=false, random uniform starting vectors are used.

For unreduced tridiagonal matrices, the default is to use Fernando starting vectors if the eigenvalues are well-separated and random uniform starting vectors otherwise. For reduced tridiagonal matrices, the default is to use random uniform starting vectors.

Further Details

TRID_INVITER uses Fernando's method for computing a first estimate of (selected) eigenvectors corresponding to (selected) approximate eigenvalues of a real n-by-n symmetric tridiagonal matrix T (by default, only for the eigenvalues which are well separated and if the input tridiagonal matrix T is unreduced). See the references (1), (2) and (4) for details.

These approximate eigenvectors are then refined (or computed if Fernando's method is not used) using an inverse iteration algorithm for all the eigenvalues at one step. See the reference (3) for details.

The eigenvectors are then orthogonalized by the Modified Gram-Schmidt or QR algorithm if clusters of eigenvalues are present, in a second step.

In a last step, the corresponding (selected) eigenvectors of the full real n-by-n symmetric matrix MAT are computed by a blocked back-transformation algorithm with the Householder transformations used to reduce the full real n-by-n symmetric matrix MAT to symmetric tridiagonal form T (see the references (5) and (6)).

Furthermore, the computation of the eigenvectors is parallelized if OPENMP is used.

TRID_INVITER may fail if some the eigenvalues specified in parameter EIGVAL are nearly identical.

For further details on Fernando's method or inverse iteration for computing eigenvectors of tridiagonal matrices or the blocked back-transformation algorithm, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(2) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

(3) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(4) **Bini, D.A., Gemignani, L., and Tisseur, F., 2005:** The Ehrlich-Aberth method for the nonsymmetric tridiagonal eigenvalue problem. SIAM J. Matrix Anal. Appl., 27, 153-175.

(5) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(6) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.4.86 subroutine trid_inviter (d, e, eigval, eigvec, failure, matp, maxiter, ortho, backward_sweep, scaling, initvec)

Purpose

TRID_INVITER computes the eigenvectors of a full real n-by-n symmetric matrix MAT, packed columnwise in a linear array MATP, corresponding to specified eigenvalues, using Fernando's method for computing (selected) eigenvectors of a real n-by-n symmetric tridiagonal matrix and inverse iteration, followed by a back-transformation procedure.

It is required that the original packed symmetric matrix MAT has been reduced to symmetric tridiagonal form T by an orthogonal similarity transformation:

Q' * MAT * Q = T

with a call to SYMTRID_CMP with parameter STORE_Q set to true, before calling TRID_INVITER.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the symmetric tridiagonal form T of MAT.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the symmetric tridiagonal form T of MAT. E(n) is arbitrary .

The size of E must verify: size(E) = size(D) = n .

EIGVAL (INPUT) real(stnd), dimension(:) On entry, selected eigenvalues of the symmetric matrix MAT. The eigenvalues must be given in decreasing order.

The size of EIGVAL must verify: size(EIGVAL) <= size(D) = n .

EIGVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed eigenvectors. The eigenvector associated with the eigenvalue EIGVAL(j) is stored in the j-th column of EIGVEC.

The shape of EIGVEC must verify:

- size(EIGVEC, 1) = size(D) = n .
- size(EIGVEC, 2) = size(EIGVAL) ,

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false indicates successful exit.
- FAILURE = true indicates that some eigenvectors failed to converge in MAXITER iterations.

MATP (INPUT) real(stnd), dimension(:) On entry, the vectors and the scalars which define the elementary reflectors used to reduce the packed real n-by-n symmetric matrix MAT to symmetric tridiagonal form T, as returned by SYMTRID_CMP with STORE_Q=true, in its argument MATP. MATP is not modified by the routine.

Back-transformation is used to find the selected eigenvectors of the original matrix MAT and these eigenvectors are stored in argument EIGVEC.

The size of MATP must verify: size(MATP) = (n * (n+1)/2)

MAXITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed in the subroutine. By default, 2 inverse iterations are performed for all the eigenvectors of the tridiagonal matrix T.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, all the eigenvectors are orthogonalized by the Modified Gram-Schmidt or QR algorithm;
- ORTHO=false, the eigenvectors are not orthogonalized by the Modified Gram-Schmidt or QR algorithm.

The default is to orthogonalize the eigenvectors only if the eigenvalues are not well-separated.

BACKWARD_SWEEP (INPUT, OPTIONAL) logical(lgl) On entry, if:

- BACKWARD_SWEEP=true and the eigenvectors are orthogonalized by the modified Gram-Schmidt algorithm, a backward sweep of the modified Gram-Schmidt algorithm is also performed;
- BACKWARD_SWEEP=false a backward sweep is not performed.

The default is not to perform a backward sweep of the modified Gram-Schmidt algorithm.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if:

- SCALING=true, the tridiagonal matrix T is scaled before computing the eigenvectors;
- SCALING=false, the tridiagonal matrix T is not scaled.

The default is to scale the tridiagonal matrix.

INITVEC (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITVEC=true, Fernando vectors are used to start the inverse iteration process;
- INITVEC=false, random uniform starting vectors are used.

For unreduced tridiagonal matrices, the default is to use Fernando starting vectors if the eigenvalues are well-separated and random uniform starting vectors otherwise. For reduced tridiagonal matrices, the default is to use random uniform starting vectors.

Further Details

TRID_INVITER uses Fernando's method for computing a first estimate of (selected) eigenvectors corresponding to (selected) approximate eigenvalues of a real n-by-n symmetric tridiagonal matrix T (by default, only for the eigenvalues which are well separated and if the input tridiagonal matrix T is unreduced). See the references (1), (2) and (4) for details.

These approximate eigenvectors are then refined (or computed if Fernando's method is not used) using an inverse iteration algorithm for all the eigenvalues at one step. See the reference (3) for details.

The eigenvectors are then orthogonalized by the Modified Gram-Schmidt or QR algorithm if clusters of eigenvalues are present in a second step.

In a final step, the corresponding (selected) eigenvectors of the full real n-by-n symmetric matrix MAT are computed by a blocked back-transformation algorithm with the Householder transformations used to reduce the full real n-by-n symmetric matrix MAT to symmetric tridiagonal form T (see the references (5) and (6)). These Householder transformations must be packed in the linear array MATP (as returned by SYMTRID_CMP) on entry of TRID_INVITER.

Furthermore, the computation of the eigenvectors is parallelized if OPENMP is used.

TRID_INVITER may fail if some the eigenvalues specified in parameter EIGVAL are nearly identical.

For further details on Fernando's method or inverse iteration for computing eigenvectors of tridiagonal matrices or the blocked back-transformation algorithm, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(2) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

(3) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(4) **Bini, D.A., Gemignani, L., and Tisseur, F., 2005:** The Ehrlich-Aberth method for the nonsymmetric tridiagonal eigenvalue problem. SIAM J. Matrix Anal. Appl., 27, 153-175.

(5) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(6) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.4.87 `subroutine gen_symtrid_mat ( type, d, e, failure, known_eigval, eigval, sort, val1, val2, l0, glu0 )`

Purpose

GEN_SYMTRID_MAT generates different types of symmetric tridiagonal matrices with known eigenvalues or specific numerical properties such as clustered eigenvalues for testing purposes of eigensolvers.

Optionally, the eigenvalues of the selected symmetric tridiagonal matrix can be computed analytically, if possible, or by a bisection algorithm with high absolute and relative accuracies.

Arguments

TYPE (INPUT) integer(i4b) Select the type of symmetric tridiagonal matrix TRID to be generated by the subroutine.

If TYPE is between 1 and 49, the subroutine generates a specific symmetric tridiagonal matrix as described in the comments inside the code of the subroutine. For other values of TYPE, all diagonal and off-diagonal elements of the symmetric tridiagonal matrix are generated from an uniform random numbers distribution between 0 and 1.

For TYPE between 1 and 17, the eigenvalues of the tridiagonal symetric matrix are known analytically. For other values of TYPE, the eigenvalues are estimated by a bisection algorithm with high accuracy.

In all cases, the eigenvalues may be output in the optional parameter EIGVAL.

D (OUTPUT) real(stnd), dimension(:) On exit, D contains the diagonal elements of the tridiagonal matrix TRID.

The size of D must verify: size(D)>=2 .

E (OUTPUT) real(stnd), dimension(:) On exit, E contains the off-diagonal elements of the tridiagonal matrix TRID. E(size(E)) is arbitrary.

The size of E must verify: size(E) = size(D) .

FAILURE (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE = false : indicates that the eigenvalues of TRID are known analytically or have been computed with high accuracy;
- FAILURE = true : indicates that the eigenvalues of TRID are not known analytically and have not been computed with maximum accuracy with the bisection algorithm.

KNOWN_EIGVAL (OUTPUT, OPTIONAL) logical(lgl) On exit:

- KNOWN_EIGVAL = true : indicates that the eigenvalues of TRID are known analytically for the selected TYPE.
- KNOWN_EIGVAL = false : indicates that the eigenvalues of TRID are not known analytically for the selected TYPE.

EIGVAL (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the eigenvalues of TRID computed analytically or estimated to high accuracy with a bisection algorithm.

The size of EIGVAL must verify: size(EIGVAL) = size(D) .

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd', if the optional argument EIGVAL is present. For other values of SORT nothing is done and EIGVAL(:) may not be sorted.

VAL1 (INPUT, OPTIONAL) real(stnd) On entry, specifies the parameter d0 for parametrized symmetric tridiagonal matrices (e.g. TYPE= 3-4, 6-10, 12, 35-38).

If this parameter is changed for TYPE between 35 and 38, which correspond to graded (or reversely graded) matrices with an arithmetic or geometric progression, care must be taken to insure that some elements of the arithmetic or geometric progression will not underflow or overflow as no checks are done in the subroutine for such errors.

The default is 1. .

VAL2 (INPUT, OPTIONAL) real(stnd) On entry, specifies the parameter e0 for parametrized symmetric tridiagonal matrices (e.g. TYPE= 3-4, 6-10, 12, 35-38).

If this parameter is changed for TYPE between 35 and 38, which correspond to graded (or reversely graded) matrices with an arithmetic or geometric progression, care must be taken to insure that some elements of the arithmetic or geometric progression will not underflow or overflow as no checks are done in the subroutine for such errors.

The default is 2. .

L0 (INPUT, OPTIONAL) integer(i4b) On entry, specify the radius of the initial matrix for parametrized form of glued tridiagonal matrices (e.g. TYPE between 45 and 49).

L0 must be greater than 0 and preferably less or equal to size(D)/2 . The default is 5. .

GLU0 (INPUT, OPTIONAL) real(stnd) On entry, specify the glue parameter for parametrized form of glued tridiagonal matrices (e.g. TYPE between 45 and 49).

The default is sqrt(epsilon(GLU0)).

Further Details

This subroutine tries to take care of imprecisions in intrinsic subroutines (e.g. like the cos function in the gfortran compiler) when computing eigenvalues by analytic formulae.

For further details on the tridiagonal matrices used for testing in GEN_SYMTRID_MAT subroutine, see:

(1) **Gladwell, G.M.L., Jones, T.H., Willms N.B., 2014:** A test matrix for an inverse eigenvalue problem. Journal of Applied Mathematics, 14, 6 pages, Article ID 515082, DOI 10.1155/2014/515082.

(2) **Clement, P.A., 1959:** A class of triple-diagonal matrices for test purposes. SIAM Review, 1(1):50-52, DOI 10.1137/1001006.

(3) **Gregory, R.T., Karney, D.L., 1969:** A collection of matrices for testing computational algorithms. New York: Wiley. Reprinted with corrections by Robert E. Krieger, Huntington, New York, 1978.

(4) **Higham, N.J., 1991: Algorithm 694:** A collection of test matrices in MATLAB. ACM Transactions on Mathematical Software 17(3):289-305 DOI 10.1145/114697.116805.

(5) **Godunov, S.K., Antonov, A.G., Kirillyuk, O.P., and Kostin, V.I., 1993:** Guaranteed Accuracy in numerical linear algebra. Kluwer Academic Publishers.

(6) **Parlett, B.N., and Vomel, C., 2005:** How the MRRR algorithm can fail on tight eigenvalue clusters. Lapack Working Note 163.

(7) **Nakatsukasa, Y., Aishima, K., and Yamazaki, I., 2012:** dqds with agressive early deflation. SIAM J. Matrix Anal. Appl., 33(1): 22-51.

(8) **Fernando, K.V., and Parlett, B.N., 1994:** Accurate singular values and differenial qd algorithms. Numer. Math., 67: 191-229.

6.5 Module_FFT_Procedures

Copyright 2024 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING FAST FOURIER TRANSFORMS.

LATEST REVISION : 26/02/2024

6.5.1 `subroutine init_fft ( shap, dim )`

Purpose

Subroutine INIT_FFT sets up constants, the Chirp functions and the Fourier transform of the Chirp functions for use by generic subroutines FFT, FFTXY, FFT_ROW and REAL_FFT for a complex valued array of shape SHAP.

Arguments

SHAP (INPUT) integer(i4b), dimension(:) Rank-one integer holding the shape of the complex valued array to be transformed. Size(SHAP) must be less or equal to 3.

DIM (INPUT, OPTIONAL) integer(i4b) Eventually specifies the index for the Fourier transform. Fourier transform on DIM-index-sections, only. DIM must be less or equal to size(SHAP).

Further Details

INIT_FFT is first called to establish and transform the Chirp functions and other constants. Then, subroutines FFT, FFTXY, FFT_ROW and REAL_FFT can be called any number of times without the precalculated constants being destroyed; a further call to INIT_FFT will only be necessary if Fourier transforms for a new length (or shape) are required.

6.5.2 subroutine init_fft (length1)

Purpose

Subroutine INIT_FFT sets up constants, the bit reverse tables, the Chirp function and the Fourier transform of the Chirp function for use by generic subroutines FFT and FFT_ROW for a series of length LENGTH1.

Arguments

LENGTH1 (INPUT) integer(i4b) The length of the complex valued sequence to be transformed. LENGTH1 may be any positive integer.

Further Details

INIT_FFT is first called to establish and transform the Chirp function and other constants. Then, subroutine FFT (or FFT_ROW) can be called any number of times without the precalculated constants being destroyed; a further call to INIT_FFT will only be necessary if a new length is required.

This subroutine is adapted from Applied Statistics algorithms AS 117 and AS 83, see:

(1) **Monro, D.M., and Branch, J.L., 1977:** The Chirp discrete Fourier transform of general length. Appl. Statist., 26 (3), 351-361.

6.5.3 subroutine init_fft (length1, length2)

Purpose

Subroutine INIT_FFT sets up constants, the Chirp functions and the Fourier transforms of the Chirp functions for use by generic subroutines FFT or FFTXY for a complex matrix of shape (LENGTH1,LENGTH2).

Arguments

LENGTH1 (INPUT) integer(i4b) The number of rows of the complex matrix to be transformed. LENGTH1 may be any positive integer.

LENGTH2 (INPUT) integer(i4b) The number of columns of the complex matrix to be transformed. LENGTH2 may be any positive integer.

Further Details

INIT_FFT is first called to establish and transform the Chirp functions and other constants. Then, subroutine FFT (or FFTXY) can be called any number of times without the precalculated constants being destroyed; a further call to INIT_FFT will only be necessary if a new shape is required.

This subroutine is adapted from Applied Statistics algorithms AS 117 and AS 83, see:

(1) **Monro, D.M., and Branch, J.L., 1977:** The Chirp discrete Fourier transform of general length. Appl. Statist., 26 (3), 351-361.

6.5.4 subroutine init_fft (length1, length2, length3)

Purpose

Subroutine INIT_FFT sets up constants, the Chirp functions and the Fourier transforms of the Chirp functions for use by generic subroutines FFT or FFTXY for a complex 3d array of shape (LENGTH1,LENGTH2,LENGTH3).

Arguments

LENGTH1 (INPUT) integer(i4b) The extent in the first dimension of the complex array to be transformed. LENGTH1 may be any positive integer.

LENGTH2 (INPUT) integer(i4b) The extent in the second dimension of the complex array to be transformed. LENGTH2 may be any positive integer.

LENGTH3 (INPUT) integer(i4b) The extent in the third dimension of the complex array to be transformed. LENGTH3 may be any positive integer.

Further Details

INIT_FFT is first called to establish and transform the Chirp functions and other constants. Then, subroutine FFT (or FFTXY) can be called any number of times without the precalculated constants being destroyed; a further call to INIT_FFT will only be necessary if a new shape is required.

This subroutine is adapted from Applied Statistics algorithms AS 117 and AS 83, see:

(1) **Monro, D.M., and Branch, J.L., 1977:** The Chirp discrete Fourier transform of general length. Appl. Statist., 26 (3), 351-361.

6.5.5 subroutine real_fft (vec, vect, forward)

Purpose

Subroutine REAL_FFT computes the Fast Fourier Transform (FFT) for a real valued sequence VEC of even length.

Arguments

VEC (INPUT) real(stnd), dimension(:) On entry, the real valued sequence to be transformed.

Size(VEC) must be an even (positive) integer.

VECT (OUTPUT) complex(stnd), dimension(:) On exit, a complex vector of length size(VEC)/2+1 containing the first size(VEC)/2+1 coefficients of the Fourier transform of the real sequence VEC. These coefficients are the positive frequency half of the full complex Fourier transform of the the real value sequence VEC.

VECT must verify: size(VECT) = size(VEC)/2 + 1.

FORWARD (INPUT) logical(lgl) Specifies whether a forward or backward Fourier transform is desired. If:

- FORWARD = true: a forward Fourier transform is computed
- FORWARD = false: a backward Fourier transform is computed.

Further Details

REAL_FFT computes the Forward Fourier Transform, VECT, according to the following formula

VECT(j+1) = [sum k=0 to nn-1] VEC(k+1) exp(- i 2 pi j k / nn)

for j=0, 1, . . . , nn/2 and where i=sqrt(-1), nn=size(VEC) and pi=3.1415923565. . .

REAL_FFT computes the Backward Fourier Transform, VECT, according to the following formula

VECT(j+1) = (1/nn) [sum k=0 to nn-1] VEC(k+1) exp(2 pi j k / nn)

for j=0, 1, . . . , nn/2 and where i=sqrt(-1), nn=size(VEC) and pi=3.1415923565. . .

The remaining values of the Fourier Transform may be computed by the following lines of code

nn = size(VEC)

nnd2 = nn/2

vect(nn:nnd2+2:-1) = conjg(vect(2:nnd2))

Before using REAL_FFT, the user must call subroutine INIT_FFT as follows :

call init_fft(size(vec)/2)

For more details on the Discrete Fourier Transform, see:

(1) **Oppenheim, A.V., and Schafer, R.W., 1999:** Discrete-Time Signal Processing. Second Edition. Prentice-Hall, New Jersey.

(2) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York.

6.5.6 subroutine real_fft (mat, matt, forward)

Purpose

Subroutine REAL_FFT computes the Fast Fourier Transform (FFT) for each row of a real valued matrix MAT.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the real valued matrix to be transformed. Size(MAT , 2) must be an even (positive) integer.

MATT (OUTPUT) complex(stnd), dimension(:,:) On exit, a complex matrix of shape size(MAT,1) by size(MAT,2)/2+1. each row of MATT contains the first size(MAT,2)/2+1 coefficients of the Fourier transform of the corresponding row of the real matrix MAT. These coefficients are the positive frequency half of the full complex Fourier transform of the corresponding row of MAT.

The shape of MATT must verify:

- size(MATT, 1) = size(MAT, 1)
- size(MATT, 2) = size(MAT, 2)/2 + 1.

FORWARD (INPUT) logical(lgl) Specifies whether a forward or backward Fourier transform is desired. If:

- FORWARD = true: a forward Fourier transform is computed
- FORWARD = false: a backward Fourier transform is computed.

Further Details

REAL_FFT computes the Forward Fourier Transform, MATT, according to the following formula

MATT(l,j+1) = [sum k=0 to nn-1] MAT(l,k+1) exp(- i 2 pi j k / nn)

for j=0, 1, . . . , nn/2 , l=1, . . . , size(MAT,1) and where i=sqrt(-1), nn=size(VEC) and pi=3.1415923565. . .

REAL_FFT computes the Backward Fourier Transform, MATT, according to the following formula

MATT(l,j+1) = (1/nn) [sum k=0 to nn-1] MAT(l,k+1) exp(2 pi j k / nn)

for j=0, 1, . . . , nn/2 , l=1, . . . , size(MAT,1) and where i=sqrt(-1), nn=size(VEC) and pi=3.1415923565. . .

The remaining values of the Fourier Transform may be computed by the following lines of code

nn = size(mat,2)

nnd2 = nn/2

matt(l,nn:nnd2+2:-1) = conjg(matt(l,2:nnd2))

Before using REAL_FFT, the user must call subroutine INIT_FFT as follows :

call init_fft((/ size(MAT,1), size(MAT,2)/2 /), dim=2)

For more details on the Discrete Fourier Transform, see:

(1) **Oppenheim, A.V., and Schafer, R.W., 1999:** Discrete-Time Signal Processing. Second Edition. Prentice-Hall, New Jersey.

(2) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York.

6.5.7 subroutine real_fft_forward (vec, vecr, veci)

Purpose

Subroutine REAL_FFT_FORWARD implements the forward discrete Fourier Transform for a real valued sequence VEC of general length.

Arguments

VEC (INPUT) real(stnd), dimension(:) On entry, the real valued sequence to be transformed.

VECR (OUTPUT) real(stnd), dimension(:) On exit, the real part of the forward discrete Fourier Transform of the sequence VEC.

VECR must verify: size(VECR) = size(VEC)/2 + 1.

VECI (OUTPUT) real(stnd), dimension(:) On exit, the imaginary part of the forward discrete Fourier Transform of the sequence VEC.

VECI must verify: size(VECI) = size(VEC)/2 + 1.

Further Details

Only, the part of the discrete Fourier Transform corresponding to the positive frequencies are computed and output in VECR and VECI.

The forward Discrete Fourier Transform is computed using Goertzel method.

For more details on the Goertzel method for computing the Discrete Fourier Transform. For further details, see:

(1) **Goertzel, G., 1958:** An Algorithm for the Evaluation of Finite Trigonometric Series, The American Mathematical Monthly, Vol. 65, No. 1, pp. 34-35

(2) **Oppenheim, A.V., and Schafer, R.W., 1999:** Discrete-Time Signal Processing. Second Edition. Prentice-Hall, New Jersey.

6.5.8 subroutine real_fft_backward (vecr, veci, vec)

Purpose

Subroutine REAL_FFT_BACKWARD implements the (real) backward discrete Fourier Transform for a complex valued sequence stored in the vector VECR (real part of the complex sequence) and VECI (imaginary part of the sequence). The resulting real discrete Fourier Transform is stored in the real vector VEC.

Size(VEC) which gives the size of the transform may be of general length.

Arguments

VECR (INPUT) real(stnd), dimension(:) On entry, the real part of the complex sequence to be transformed.

VECR must verify: size(VECR) = size(VEC)/2 + 1.

VECI (INPUT) real(stnd), dimension(:) On entry, the imaginary part of the complex sequence to be transformed.

VECI must verify: size(VECI) = size(VEC)/2 + 1.

VEC (OUTPUT) real(stnd), dimension(:) On exit, the discrete Fourier transform real valued sequence.

Further Details

The backward Discrete Fourier Transform is computed using Goertzel method.

For more details on the Goertzel method for computing the Discrete Fourier Transform. For further details, see:

(1) **Goertzel, G., 1958:** An Algorithm for the Evaluation of Finite Trigonometric Series, The American Mathematical Monthly, Vol. 65, No. 1, pp. 34-35

(2) **Oppenheim, A.V., and Schafer, R.W., 1999:** Discrete-Time Signal Processing. Second Edition. Prentice-Hall, New Jersey.

6.5.9 subroutine real_fft_forward (mat, matr, mati, dim)

Purpose

Subroutine REAL_FFT_FORWARD computes the forward discrete Fourier Transform of each row (DIM=2) or each column (DIM=1) of the real matrix MAT.

Size(MAT,DIM) which gives the size of the transform may be of general length.

The real parts of the forward discrete Fourier Transforms are stored (rowwise) in MATR and the corresponding imaginary parts of the transforms are stored in MATI.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the real valued sequences to be transformed.

MATR (OUTPUT) real(stnd), dimension(:,:) On exit, the real part of the forward discrete Fourier Transform of the sequences stored in MAT.

The shape of MATR must verify:

- size(MATR, 1) = size(MAT, 3-DIM)
- size(MATR, 2) = size(MAT, DIM)/2 + 1.

MATI (OUTPUT) real(stnd), dimension(:,:) On exit, the imaginary part of the forward discrete Fourier Transform of the sequences stored in MAT.

The shape of MATI must verify:

- size(MATI, 1) = size(MAT, 3-DIM)
- size(MATI, 2) = size(MAT, DIM)/2 + 1.

DIM (INPUT) integer(i4b) Specifies the index for the Fourier transform. If:

- DIM = 1 : Fourier transform on first index.
- DIM = 2 : Fourier transform on second index.

Further Details

Only, the parts of the discrete Fourier Transforms corresponding to the positive frequencies are computed and output in MATR and MATI.

The forward Discrete Fourier Transform is computed using Goertzel method.

For more details on the Goertzel method for computing the Discrete Fourier Transform. For further details, see:

(1) **Goertzel, G., 1958:** An Algorithm for the Evaluation of Finite Trigonometric Series, The American Mathematical Monthly, Vol. 65, No. 1, pp. 34-35

(2) **Oppenheim, A.V., and Schafer, R.W., 1999:** Discrete-Time Signal Processing. Second Edition. Prentice-Hall, New Jersey.

6.5.10 `subroutine real_fft_backward ( matr, mati, mat, dim )`

Purpose

Subroutine REAL_FFT_BACKWARD computes the (real) backward discrete Fourier Transform for complex valued sequences stored in the matrices MATR (real part of the sequences stored rowwise) and MATI (imaginary part of the sequences stored rowwise). The resulting real discrete Fourier Transforms are stored in the rows (DIM=2) or the columns (DIM=1) of the real matrix MAT.

Size(MAT,DIM) which gives the size of the transform may be of general length.

Arguments

MATR (INPUT) real(stnd), dimension(:,:) On entry, the real part of the complex sequences to be transformed.

The shape of MATR must verify:

- size(MATR, 1) = size(MAT, 3-DIM)
- size(MATR, 2) = size(MAT, DIM)/2 + 1.

MATI (INPUT) real(stnd), dimension(:,:) On entry, the imaginary part of the complex sequences to be transformed.

The shape of MATI must verify:

- size(MATI, 1) = size(MAT, 3-DIM)
- size(MATI, 2) = size(MAT, DIM)/2 + 1.

MAT (OUTPUT) real(stnd), dimension(:,:) On exit, the real backward discrete Fourier transforms of the complex sequences stored rowwise in MATR and MATI.

DIM (INPUT) integer(i4b) Specifies the index for the Fourier transform. If:

- DIM = 1 : Fourier transform on first index.
- DIM = 2 : Fourier transform on second index.

Further Details

The backward Discrete Fourier Transform is computed using Goertzel method.

For more details on the Goertzel method for computing the Discrete Fourier Transform. For further details, see:

(1) **Goertzel, G., 1958:** An Algorithm for the Evaluation of Finite Trigonometric Series, The American Mathematical Monthly, Vol. 65, No. 1, pp. 34-35

(2) **Oppenheim, A.V., and Schafer, R.W., 1999:** Discrete-Time Signal Processing. Second Edition. Prentice-Hall, New Jersey.

6.5.11 `subroutine fftxy ( x, y, fftx, ffty)`

Purpose

Given two real valued sequences of the same length, X and Y, FFTXY returns the Fast Fourier Transforms of these sequences in the two complex valued sequences FFTX and FFTY.

Arguments

X, Y (INPUT) real(stnd), dimension(:) On entry, the real valued sequences to be transformed.

X and Y must verify: size(X) = size(Y).

FFTX, FFTY (OUTPUT) complex(stnd), dimension(:) On exit, FFTX, FFTY are replaced by the Fourier transforms of X and Y, respectively.

FFTX and FFTY must verify: size(FFTX) = size(FFTY) = size(X) = size(Y).

Further Details

Size(FFTX) = size(FFTY) = size(X) = size(Y) may be of general length.

If size(X) is an exact power of two, Bailey's Four-Step FFT algorithm is used, otherwise the CHIRP-Z transform is employed.

Before using FFTXY, the user must call subroutine INIT_FFT as follows :

```
call init_fft( size(X) )
```

This subroutine is adapted from Applied Statistics algorithms AS 117 and AS 83 and Bailey (1990). For more details, see:

(1) **Monro, D.M., and Branch, J.L., 1977:** The Chirp discrete Fourier transform of general length. Appl. Statist., 26 (3), 351-361.

(2) **Bailey, D., 1990:** FFTs in External or Hierarchical Memory, The Journal of Supercomputing, 4, 23-35.

6.5.12 `subroutine fftxy ( x, y, fftx, ffty )`

Purpose

Given two real valued matrices of the same shape, X and Y, FFTXY returns the Fast Fourier Transforms of X and Y in the two complex valued matrices FFTX and FFTY, respectively.

Arguments

X, Y (INPUT) real(stnd), dimension(:,:) On entry, the real valued matrices to be transformed.

X and Y must verify the equality: shape(X) = shape(Y).

FFTX, FFTY (OUTPUT) complex(stnd), dimension(:,:) On exit, FFTX and FFTY are replaced by the Fourier transforms of X and Y, respectively.

FFTX and FFTY must verify the equalities:

- shape(FFTX) = shape(FFTY) = shape(X) = shape(Y).

Further Details

Depending if size(X,1) and size(X,2) are exact powers of two or not, a radix-2 decimation-in-time Cooley-Tukey algorithm or a CHIRP-Z transform is employed.

Before using FFTXY, the user must call subroutine INIT_FFT as follows :

```
call init_fft( size(X,1), size(X,2) )
```

For more details, see:

(1) **Monro, D.M., and Branch, J.L., 1977:** The Chirp discrete Fourier transform of general length. Appl. Statist., 26 (3), 351-361.

(2) **Cooley, J.W., Lewis, P., and Welch, P., 1969:** The Fast Fourier Transform and its Applications. IEEE Trans on Education, 12, 1, 28-34.

(3) **Oppenheim, A.V., and Schafer, R.W., 1999:** Discrete-Time Signal Processing. Second Edition. Prentice-Hall, New Jersey.

6.5.13 subroutine fftxy (x, y, fftx, ffty)

Purpose

Given two real valued 3D arrays of the same shape, X and Y, FFTXY returns the Fast Fourier Transforms of X and Y in the two complex valued 3D arrays FFTX and FFTY, respectively.

Arguments

X, Y (INPUT) real(stnd), dimension(:,:,:) On entry, the real valued 3D arrays to be transformed.

X and Y must verify: shape(X) = shape(Y).

FFTX, FFTY (OUTPUT) complex(stnd), dimension(:,:,:) On exit, FFTX and FFTY are replaced by the Fourier transforms of X and Y, respectively.

FFTX and FFTY must verify: shape(FFTX) = shape(FFTY) = shape(X) = shape(Y).

Further Details

Depending if size(X,1), size(X,2) and size(X,3) are exact powers of two or not, a radix-2 decimation-in-time Cooley-Tukey algorithm or a CHIRP-Z transform is employed.

Before using FFTXY, the user must call subroutine INIT_FFT as follows :

call init_fft(size(X,1), size(X,2), size(X,3))

For more details, see:

(1) **Monro, D.M., and Branch, J.L., 1977:** The Chirp discrete Fourier transform of general length. Appl. Statist., 26 (3), 351-361.

(2) **Cooley, J.W., Lewis, P., and Welch, P., 1969:** The Fast Fourier Transform and its Applications. IEEE Trans on Education, 12, 1, 28-34.

(3) **Oppenheim, A.V., and Schafer, R.W., 1999:** Discrete-Time Signal Processing. Second Edition. Prentice-Hall, New Jersey.

6.5.14 subroutine fftxy (x, y, fftx, ffty, dim)

Purpose

Given two real valued matrices of the same shape, X and Y, FFTXY returns the Fast Fourier Transforms of the rows (DIM=2) or the columns (DIM=1) of X and Y in the two complex valued matrices FFTX and FFTY, respectively.

Arguments

X, Y (INPUT) real(stnd), dimension(:,:) On entry, the real valued matrices to be transformed.

X and Y must verify: shape(X) = shape(Y).

FFTX, FFTY (OUTPUT) complex(stnd), dimension(:,:) On exit:

- each row of FFTX and FFTY are replaced by the Fourier transforms of the rows of X and Y, respectively, if DIM=2 .

- each column of FFTX and FFTY are replaced by the Fourier transforms of the colums of X and Y, respectively, if DIM=1 .

FFTX and FFTY must verify: shape(FFTX) = shape(FFTY) = shape(X) = shape(Y).

DIM (INPUT) integer(i4b) Specifies the index for the Fourier transform. If:

- DIM = 1 : Fourier transform on first index,
- DIM = 2 : Fourier transform on second index.

Further Details

Size(FFTX,DIM) = size(FFTY,DIM) = size(X,DIM) = size(Y,DIM) may be of general length.

Before using FFTXY, the user must call subroutine INIT_FFT as follows :

call init_fft((/ size(X,1), size(X,2) /), dim=DIM)

6.5.15 `subroutine fftxy ( x, y, fftx, ffty, dim )`

Purpose

Given two real valued 3D arrays of the same shape, X and Y, FFTXY returns the Fast Fourier Transforms of each DIM-index section of X and Y in the two complex valued 3D arrays FFTX and FFTY, respectively.

Arguments

X, Y (INPUT) real(stnd), dimension(:,:,:) On entry, the real valued 3D arrays to be transformed.

X and Y must verify: shape(X) = shape(Y).

FFTX, FFTY (OUTPUT) complex(stnd), dimension(:,:,:) On exit, the DIM-index sections of FFTX and FFTY are replaced by the Fourier transforms of the DIM-index sections of X and Y, respectively.

FFTX and FFTY must verify: shape(FFTX) = shape(FFTY) = shape(X) = shape(Y).

DIM (INPUT) integer(i4b) Specifies the index for the Fourier transform. If:

- DIM = 1 : Fourier transform on first-index-sections.
- DIM = 2 : Fourier transform on second-index-sections.
- DIM = 3 : Fourier transform on third-index-sections.

Further Details

Size(FFTX,DIM) = size(FFTY,DIM) = size(X,DIM) = size(Y,DIM) may be of general length.

Before using FFTXY, the user must call subroutine INIT_FFT as follows:

call init_fft((/ size(X,1), size(X,2), size(X,3) /), dim=DIM)

6.5.16 subroutine fft (dat, forward)

Purpose

Subroutine FFT implements the Fast Fourier Transform for a complex valued sequence DAT of general length.

Forward discrete Fourier transform of a vector DAT(:) is given by

t(DAT)(j) = [sum k=0 to nn-1] DAT(k) exp(- i 2 pi j k / nn)

Backward discrete Fourier transform of a vector DAT(:) is given by

t(DAT)(j) = (1/nn) [sum k=0 to nn-1] DAT(k) exp(i 2 pi j k / nn)

where i = sqrt(-1), nn = size(DAT) and pi = 3.1415923565. . .

Note that the indexing of DAT is shifted by one : DAT(0) stored in DAT(1), . . . , DAT(nn-1) stored in DAT(nn).

Arguments

DAT (INPUT/OUTPUT) complex(stnd), dimension(:) On entry, the complex valued sequence to be transformed. On exit, DAT is replaced by the Fourier transform.

FORWARD (INPUT) logical(lgl) Specifies whether a forward or backward Fourier transform is desired. If:

- FORWARD = true: a forward Fourier transform is computed
- FORWARD = false: a backward Fourier transform is computed.

Further Details

If size(DAT) is an exact power of two, Bailey's Four-Step FFT algorithm is used, otherwise the CHIRP-Z transform is employed.

Before using FFT, the user must call subroutine INIT_FFT as follows :

```
call init_fft( size(DAT) )
```

This subroutine is adapted from Applied Statistics algorithms AS 117 and AS 83 and Bailey (1990). For more details, see:

(1) **Monro, D.M., and Branch, J.L., 1977:** The Chirp discrete Fourier transform of general length. Appl. Statist., 26 (3), 351-361.

(2) **Bailey, D., 1990:** FFTs in External or Hierarchical Memory. The Journal of Supercomputing, 4, 23-35.

6.5.17 subroutine fft (dat, forward)

Purpose

Subroutine FFT implements the Fast Fourier Transform for a complex matrix DAT of general shape.

Arguments

DAT (INPUT/OUTPUT) complex(stnd), dimension(:,:) On entry, the complex matrix to be transformed. On exit, DAT is replaced by its Fourier transform.

FORWARD (INPUT) logical(lgl) Specifies whether a forward or backward Fourier transform is desired. If:

- FORWARD = true: a forward Fourier transform is computed
- FORWARD = false: a backward Fourier transform is computed.

Further Details

Depending if size(DAT,1) and size(DAT,2) are exact powers of two or not, a radix-2 decimation-in-time Cooley-Tukey algorithm or a CHIRP-Z transform is employed.

Before using FFT, the user must call subroutine INIT_FFT as follows :

```
call init_fft( (/ size(DAT,1), size(DAT,2) /) )
```

For more details, see:

(1) **Monro, D.M., and Branch, J.L., 1977:** The Chirp discrete Fourier transform of general length. Appl. Statist., 26 (3), 351-361.

(2) **Cooley, J.W., Lewis, P., and Welch, P., 1969:** The Fast Fourier Transform and its Applications. IEEE Trans on Education, 12, 1, 28-34.

(3) **Oppenheim, A.V., and Schafer, R.W., 1999:** Discrete-Time Signal Processing. Second Edition. Prentice-Hall, New Jersey.

6.5.18 `subroutine fft ( dat, forward)`

Purpose

Subroutine FFT implements the Fast Fourier Transform for a complex 3D array DAT of general shape.

Arguments

DAT (INPUT/OUTPUT) complex(stnd), dimension(:,:,:) On entry, the complex array to be transformed. On exit, DAT is replaced by its Fourier transform.

FORWARD (INPUT) logical(lgl) Specifies whether a forward or backward Fourier transform is desired. If

- FORWARD = true: a forward Fourier transform is computed
- FORWARD = false: a backward Fourier transform is computed.

Further Details

Depending if size(DAT,1), size(DAT,2) and size(DAT,3) are exact powers of two or not, a radix-2 decimation-in-time Cooley-Tukey algorithm or a CHIRP-Z transform is employed.

Before using FFT, the user must call subroutine INIT_FFT as follows :

```
call init_fft( (/ size(DAT,1), size(DAT,2), size(DAT,3) /) )
```

For more details, see:

(1) **Monro, D.M., and Branch, J.L., 1977:** The Chirp discrete Fourier transform of general length. Appl. Statist., 26 (3), 351-361.

(2) **Cooley, J.W., Lewis, P., and Welch, P., 1969:** The Fast Fourier Transform and its Applications. IEEE Trans on Education, 12, 1, 28-34.

(3) **Oppenheim, A.V., and Schafer, R.W., 1999:** Discrete-Time Signal Processing. Second Edition. Prentice-Hall, New Jersey.

6.5.19 `subroutine fft ( dat, forward, dim)`

Purpose

Subroutine FFT replaces each row of DAT by its Fourier transform. (DIM=2) or each column of DAT by its Fourier transform (DIM=1). Size(DAT,DIM) may be of general length.

Arguments

DAT (INPUT/OUTPUT) complex(stnd), dimension(:,:) On entry, the complex valued sequences to be transformed. On exit, each row of DAT is replaced by its Fourier transform if DIM=2 or each column of DAT is replaced by its Fourier transform if DIM=1.

FORWARD (INPUT) logical(lgl) Specifies whether a forward or backward Fourier transform is desired. If:

- FORWARD = true: a forward Fourier transform is computed
- FORWARD = false: a backward Fourier transform is computed.

DIM (INPUT) integer(i4b) Specifies the index for the Fourier transform. If:

- DIM = 1 : Fourier transform on first index,
- DIM = 2 : Fourier transform on second index.

Further Details

If size(DAT,DIM) is an exact power of two, a 1D in-place complex-complex radix-2 decimation-in-time Cooley-Tukey FFT algorithm is used, otherwise the CHIRP-Z transform is employed.

Before using FFT, the user must call subroutine INIT_FFT as follows :

```
call init_fft( (/ size(DAT,1), size(DAT,2) /), dim=DIM )
```

This subroutine is adapted from Applied Statistics algorithms AS 117 and AS 83, see:

(1) **Monro, D.M., and Branch, J.L., 1977:** The Chirp discrete Fourier transform of general length. Appl. Statist., 26 (3), 351-361.

6.5.20 `subroutine fft ( dat, forward, dim)`

Purpose

Subroutine FFT replaces each DIM-index section of DAT by its Fourier transform. Size(DAT,DIM) may be of general length.

Arguments

DAT (INPUT/OUTPUT) complex(stnd), dimension(:,:,:) On entry, the complex valued sequences to be transformed. On exit, the DIM-index sections of DAT are replaced by their Fourier transforms.

FORWARD (INPUT) logical(lgl) Specifies whether a forward or backward Fourier transform is desired. If:

- FORWARD = true: a forward Fourier transform is computed
- FORWARD = false: a backward Fourier transform is computed.

DIM (INPUT) integer(i4b) Specifies the index for the Fourier transform. If:

- DIM = 1 : Fourier transform on first-index-sections,
- DIM = 2 : Fourier transform on second-index-sections,
- DIM = 3 : Fourier transform on third-index-sections.

Further Details

If size(DAT,DIM) is an exact power of two, a 1D in-place complex-complex radix-2 decimation-in-time Cooley-Tukey FFT algorithm is used, otherwise the CHIRP-Z transform is employed.

Before using FFT_DIM_CT, the user must call subroutine INIT_FFT as follows :

call init_fft((/ size(DAT,1), size(DAT,2), size(DAT,3) /), dim=DIM)

This subroutine is adapted from Applied Statistics algorithms AS 117 and AS 83, see:

(1) **Monro, D.M., and Branch, J.L., 1977:** The Chirp discrete Fourier transform of general length. Appl. Statist., 26 (3), 351-361.

6.5.21 `subroutine fft_row ( dat, forward)`

Purpose

Subroutine FFT_ROW implements the Fast Fourier Transform for a complex valued sequence DAT of general length.

Forward discrete Fourier transform of a vector DAT(:) is given by

t(DAT)(j) = [sum k=0 to nn-1] DAT(k) exp(- i 2 pi j k / nn)

Backward discrete Fourier transform of a vector DAT(:) is given by

t(DAT)(j) = (1/nn) [sum k=0 to nn-1] DAT(k) exp(i 2 pi j k / nn)

where i = sqrt(-1), nn = size(DAT) and pi = 3.1415923565. . .

Note that the indexing of DAT is shifted by one : DAT(0) stored in DAT(1), . . . , DAT(nn-1) stored in DAT(nn).

Arguments

DAT (INPUT/OUTPUT) complex(stnd), dimension(:) On entry, the complex valued sequence to be transformed. On exit, DAT is replaced by the Fourier transform.

FORWARD (INPUT) logical(lgl) Specifies whether a forward or backward Fourier transform is desired. If:

- FORWARD = true: a forward Fourier transform is computed
- FORWARD = false: a backward Fourier transform is computed.

Further Details

If size(DAT) is an exact power of two, Bailey's Four-Step FFT algorithm is used, otherwise the CHIRP-Z transform is employed.

This is the parallelized version of FFT subroutine (Parallelization is done with OPENMP directives).

Before using FFT_ROW, the user must call subroutine INIT_FFT as follows :

```
call init_fft( size(DAT) )
```

This subroutine is adapted from Applied Statistics algorithms AS 117 and AS 83. For more details, see:

(1) **Monro, D.M., and Branch, J.L., 1977:** The Chirp discrete Fourier transform of general length. Appl. Statist., 26 (3), 351-361.

(2) **Bailey, D., 1990:** FFTs in External or Hierarchical Memory. The Journal of Supercomputing, 4, 23-35.

6.5.22 `subroutine fft_row ( dat, forward)`

Purpose

Subroutine FFT_ROW replaces each row of DAT by its Fourier transform. Size(DAT,2) may be of general length.

Arguments

DAT (INPUT/OUTPUT) complex(stnd), dimension(:,:) On entry, the complex valued sequences to be transformed. On exit, each row of DAT is replaced by its Fourier transform.

FORWARD (INPUT) logical(lgl) Specifies whether a forward or backward Fourier transform is desired. If:

- FORWARD = true: a forward Fourier transform is computed
- FORWARD = false: a backward Fourier transform is computed.

Further Details

If size(DAT,2) is an exact power of two, a 1D complex-complex radix-2 decimation-in-time Cooley-Tukey FFT algorithm is used, otherwise the CHIRP-Z transform is employed.

This is a parallelized FFT subroutine if OPENMP is used (Parallelization is done with OPENMP directives).

Before using FFT_ROW, the user must call subroutine INIT_FFT as follows :

```
call init_fft( (/ size(DAT,1), size(DAT,2) /), dim=2 )
```

This subroutine is adapted from Applied Statistics algorithms AS 117 and AS 83. For further details, see:

(1) **Monro, D.M., and Branch, J.L., 1977:** The Chirp discrete Fourier transform of general length. Appl. Statist., 26 (3), 351-361.

6.5.23 subroutine end_fft ()

Purpose

END_FFT deallocates the workspace previously allocated by a call to INIT_FFT.

Arguments

None

6.6 Module_Giv_Procedures

Copyright 2024 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING GIVENS TOOLS (REFLECTIONS AND ROTATIONS).

LATEST REVISION : 11/03/2024

6.6.1 subroutine define_rot_givens (a, b, cs, sn)

Purpose

DEFINE_ROT_GIVENS generates the cosine and sine of a Givens plane rotation, ROT, so that

(A B) ROT = (R 0)

where R >= 0 and ROT is 2-by-2 matrix defined by

(+CS -SN)

(+SN +CS)

with CS**(2) + SN**(2) = 1.

Arguments

A (INPUT) real(stnd) The first component of vector to be rotated.

B (INPUT) real(stnd) The second component of vector to be rotated.

CS (OUTPUT) real(stnd) The cosine of the rotation.

SN (OUTPUT) real(stnd) The sine of the rotation.

Further Details

A and B are unchanged on return.

Normally, the subprogram APPLY_ROT_GIVENS(VECA, VECB, CS, SN) will next be called to apply the rotation to a n-by-2 matrix [VECA VECB].

6.6.2 `subroutine rot_givens ( a, b )`

Purpose

ROT_GIVENS applies a Givens plane rotation, ROT, so that

(A B) ROT = (R 0)

where ROT is 2-by-2 matrix defined by

(+CS -SN)

(+SN +CS)

with CS**(2) + SN**(2) = 1.

On output, the rotation is also stored in compact form in B.

Arguments

A (INPUT/OUTPUT) real(stnd) The first component of vector to be rotated.

On output, R = (+/-)sqrt(A**(2) + B**(2)) overwrites A.

B (INPUT/OUTPUT) real(stnd) The second component of vector to be rotated.

On output, Z overwrites B. Z allows CS and SN to be recovered by the following algorithm:

- If Z = 1 set CS = 0 and SN = 1
- If abs(Z) < 1 set SN = Z and CS = sqrt(1 - SN**(2))
- If abs(Z) > 1 set CS = 1/Z and SN = sqrt(1 - CS**(2))

Further Details

Normally, the subprogram APPLY_ROT_GIVENS(VECA, VECB, B) will next be called to apply the rotation to a n-by-2 matrix [VECA VECB].

6.6.3 subroutine rot_givens (a, b, cs, sn)

Purpose

ROT_GIVENS generates and applies a Givens plane rotation, ROT, so that

(A B) ROT = (R 0)

where ROT is 2-by-2 matrix defined by

(+CS -SN)

(+SN +CS)

with CS**(2) + SN**(2) = 1.

On output, the rotation is also stored in compact form in B.

Arguments

A (INPUT/OUTPUT) real(stnd) The first component of vector to be rotated.

On output, R = (+/-)sqrt(A**(2) + B**(2)) overwrites A.

B (INPUT/OUTPUT) real(stnd) The second component of vector to be rotated.

On output, Z overwrites B. Z allows CS and SN to be recovered by the following algorithm:

- If Z = 1 set CS = 0 and SN = 1
- If abs(Z) < 1 set SN = Z and CS = sqrt(1 - SN**(2))
- If abs(Z) > 1 set CS = 1/Z and SN = sqrt(1 - CS**(2))

CS (OUTPUT) real(stnd) The cosine of the rotation.

SN (OUTPUT) real(stnd) The sine of the rotation.

Further Details

Normally, the subprograms APPLY_ROT_GIVENS(VECA, VECB, CS, SN) or APPLY_ROT_GIVENS(VECA, VECB, B) will next be called to apply the rotation to a n-by-2 matrix [VECA VECB].

6.6.4 subroutine rot_givens (veca, vecb)

Purpose

ROT_GIVENS applies a Givens plane rotation, ROT, to the n-by-2 matrix [VECA VECB]. The rotation is designed to annilhate the first element of VECB (e.g. VECB(1)). That is,

[VECA VECB] ROT = [(CS*VECA + SN*VECB) (-SN*VECA + CS*VECB)]

where

- CS**(2) + SN**(2) = 1,
- -SN*VECA(1) + CS*VECB(1) = 0
- and ROT is a 2-by-2 matrix defined by

(+CS -SN)

(+SN +CS)

On output, the rotation is also stored in compact form in VECB(1).

Arguments

VECA (INPUT/OUTPUT) real(stnd), dimension(:) The first vector to be rotated.

On output, CS*VECA + SN*VECB overwrites VECA.

VECB (INPUT/OUTPUT) real(stnd), dimension(:) The second vector to be rotated.

On output, -SN*VECA(2:) + CS*VECB(2:) overwrites VECB(2:) and Z overwrites VECB(1). Z allows CS and SN to be recovered by the following algorithm:

- If Z = 1 set CS = 0 and SN = 1
- If abs(Z) < 1 set SN = Z and CS = sqrt(1 - SN**(2))
- If abs(Z) > 1 set CS = 1/Z and SN = sqrt(1 - CS**(2))

Further Details

It is assumed that VECA and VECB have the same size.

The subprograms APPLY_ROT_GIVENS(VECC, VECD, VECB(1)) may next be called to apply the rotation to another n-by-2 matrix [VECC VECD].

6.6.5 `subroutine rot_givens ( veca, vecb, cs, sn )`

Purpose

ROT_GIVENS defines and applies a Givens plane rotation, ROT, to the n-by-2 matrix [VECA VECB]. The rotation is designed to annilhate the first element of VECB (e.g. VECB(1)). That is,

[VECA VECB] ROT = [(CS*VECA + SN*VECB) (-SN*VECA + CS*VECB)]

where

- CS**(2) + SN**(2) = 1,
- -SN*VECA(1) + CS*VECB(1) = 0
- and ROT is a 2-by-2 matrix defined by

  (+CS -SN)

  (+SN +CS)

On output, the rotation is also stored in compact form in VECB(1).

Arguments

VECA (INPUT/OUTPUT) real(stnd), dimension(:) The first vector to be rotated.

On output, CS*VECA + SN*VECB overwrites VECA.

VECB (INPUT/OUTPUT) real(stnd), dimension(:) The second vector to be rotated.

On output, -SN*VECA(2:) + CS*VECB(2:) overwrites VECB(2:) and Z overwrites VECB(1). Z allows CS and SN to be recovered by the following algorithm:

- If Z = 1 set CS = 0 and SN = 1
- If abs(Z) < 1 set SN = Z and CS = sqrt(1 - SN**(2))
- If abs(Z) > 1 set CS = 1/Z and SN = sqrt(1 - CS**(2))

CS (OUTPUT) real(stnd) The cosine of the rotation.

SN (OUTPUT) real(stnd) The sine of the rotation.

Further Details

It is assumed that VECA and VECB have the same size.

Normally, the subprograms APPLY_ROT_GIVENS(VECC, VECD, CS, SN) or APPLY_ROT_GIVENS(VECC, VECD, VECB(1)) will next be called to apply the rotation to a n-by-2 matrix [VECC VECD].

6.6.6 subroutine apply_rot_givens (c, d, b)

Purpose

APPLY_ROT_GIVENS reconstructs and applies a Givens plane rotation, ROT, stored in compact form in B, to the vector (C D).

That is, the value B allows the cosine and sine of the Givens plane rotation to be recovered by the following algorithm:

- If B = 1 set CS = 0 and SN = 1
- If abs(B) < 1 set SN = B and CS = sqrt(1 - SN**(2))
- If abs(B) > 1 set CS = 1/B and SN = sqrt(1 - CS**(2))

Next, the Givens plane rotation, ROT, is applied to the vector (C D) :

(C D) ROT = ((CS*C + SN*D) (-SN*C + CS*D))

where ROT is a 2-by-2 matrix defined by

(+CS -SN)

(+SN +CS)

Arguments

C (INPUT/OUTPUT) real(stnd) The first element of vector to be rotated.

On output, CS*C + SN*D overwrites C.

D (INPUT/OUTPUT) real(stnd) The second element of vector to be rotated.

On output, -SN*C + CS*D overwrites D.

B (INPUT) real(stnd) The real number, which allows the cosine and sine of the Givens plane rotation to be recovered.

Further Details

Normally:

- the subprogram APPLY_ROT_GIVENS(C, D, B) is called to apply a Givens rotation to the vector (C D) after a call to ROT_GIVENS(A, B, CS, SN) or ROT_GIVENS(A, B).
- the subprogram APPLY_ROT_GIVENS(C, D, VECB(1)) is called to apply a Givens rotation to the vector (C D) after a call to ROT_GIVENS(VECA, VECB, CS, SN) or ROT_GIVENS(VECA, VECB) where VECA and VECB are two vectors of the same length.

6.6.7 `subroutine apply_rot_givens ( c, d, cs, sn )`

Purpose

APPLY_ROT_GIVENS applies a Givens plane rotation, ROT, to to the vector (C D). That is,

(C D) ROT = ((CS*C + SN*D) (-SN*C + CS*D))

where ROT is a 2-by-2 matrix defined by

(+CS -SN)

(+SN +CS)

Arguments

C (INPUT/OUTPUT) real(stnd) The first element of vector to be rotated.

On output, CS*C + SN*D overwrites C.

D (INPUT/OUTPUT) real(stnd) The second element of vector to be rotated.

On output, -SN*C + CS*D overwrites D.

CS (INPUT) real(stnd) The cosine of the rotation.

SN (INPUT) real(stnd) The sine of the rotation.

Further Details

Normally, the subprogram APPLY_ROT_GIVENS(C, D, CS, SN) is called to apply a Givens rotation to the vector (C D) after a call to DEFINE_ROT_GIVENS(A, B, CS, SN), ROT_GIVENS(A, B, CS, SN) or ROT_GIVENS(VECA, VECB, CS, SN).

6.6.8 `subroutine apply_rot_givens ( vecc, vecd, b )`

Purpose

APPLY_ROT_GIVENS reconstructs and applies a Givens plane rotation, ROT, stored in compact form in B, to the n-by-2 matrix [VECC VECD].

That is, the value B allows the cosine and sine of the Givens plane rotation to be recovered by the following algorithm:

- If B = 1 set CS = 0 and SN = 1

- If abs(B) < 1 set SN = B and CS = sqrt(1 - SN**(2))
- If abs(B) > 1 set CS = 1/B and SN = sqrt(1 - CS**(2))

Next, the Givens plane rotation, ROT, is applied to the n-by-2 matrix [VECC VECD]:

[VECC VECD] ROT = [(CS*VECC + SN*VECD) (-SN*VECC + CS*VECD)]

where ROT is a 2-by-2 matrix defined by

(+CS -SN)

(+SN +CS)

Arguments

VECC (INPUT/OUTPUT) real(stnd), dimension(:) The first vector to be rotated.

On output, CS*VECC + SN*VECD overwrites VECC.

VECD (INPUT/OUTPUT) real(stnd), dimension(:) The second vector to be rotated.

On output, -SN*VECC + CS*VECD overwrites VECD.

B (INPUT) real(stnd) The real number which allows the cosine and sine of the Givens plane rotation to be recovered.

Further Details

Normally, the subprogram APPLY_ROT_GIVENS(VECC, VECD, B) is called to apply a Givens rotation to the n-by-2 matrix [VECC VECD] after a call to ROT_GIVENS(A, B, CS, SN) or ROT_GIVENS(A, B).

Normally, the subprogram APPLY_ROT_GIVENS(VECC, VECD, VECB(1)) is called to apply a Givens rotation to the n-by-2 matrix [VECC VECD] after a call to ROT_GIVENS(VECA, VECB, CS,SN) or ROT_GIVENS(VECA, VECB) where VECA and VECB are two vectors of the same length.

It is assumed that VECC and VECD have the same size.

6.6.9 `subroutine apply_rot_givens ( vecc, vecd, cs, sn )`

Purpose

APPLY_ROT_GIVENS applies a Givens plane rotation, ROT, to the n-by-2 matrix [VECC VECD]. That is,

[VECC VECD] ROT = [(CS*VECC + SN*VECD) (-SN*VECC + CS*VECD)]

where ROT is a 2-by-2 matrix defined by

(+CS -SN)

(+SN +CS)

Arguments

VECC (INPUT/OUTPUT) real(stnd), dimension(:) The first vector to be rotated.

On output, CS*VECC + SN*VECD overwrites VECC.

VECD (INPUT/OUTPUT) real(stnd), dimension(:) The second vector to be rotated.

On output, -SN*VECC + CS*VECD overwrites VECD.

CS (INPUT) real(stnd) The cosine of the rotation.

SN (INPUT) real(stnd) The sine of the rotation.

Further Details

Normally, the subprogram APPLY_ROT_GIVENS(VECC, VECD, CS, SN) is called to apply a Givens rotation to the n-by-2 matrix [VECC VECD] after a call to DEFINE_ROT_GIVENS(A, B, CS, SN), ROT_GIVENS(A, B, CS, SN) or ROT_GIVENS(VECA, VECB, CS, SN).

It is assumed that VECC and VECD have the same size.

6.6.10 `subroutine givens_vec ( veca, vecb )`

Purpose

GIVENS defines and applies a Givens plane rotation, ROT, to the n-by-2 matrix [VECA VECB]. The rotation is designed to annilhate the first element of VECB (e.g. VECB(1)). That is,

[VECA VECB] ROT = [(CS*VECA + SN*VECB) (-SN*VECA + CS*VECB)]

where:

- CS**(2) + SN**(2) = 1,
- -SN*VECA(1) + CS*VECB(1) = 0,
- CS*VECA(1) + SN*VECB(1) >= 0.

and ROT is a 2-by-2 matrix defined by

(+CS -SN)

(+SN +CS)

Arguments

VECA (INPUT/OUTPUT) real(stnd), dimension(:) The first vector to be rotated.

On output, CS*VECA + SN*VECB overwrites VECA.

VECB (INPUT/OUTPUT) real(stnd), dimension(:) The second vector to be rotated.

On output, -SN*VECA + CS*VECB overwrites VECB.

Further Details

It is assumed that VECA and VECB have the same size.

6.6.11 subroutine givens_vec (veca, vecb, cs, sn)

Purpose

GIVENS defines and applies a Givens plane rotation, ROT, to the n-by-2 matrix [VECA VECB]. The rotation is designed to annilhate the first element of VECB (e.g. VECB(1)). That is,

[VECA VECB] ROT = [(CS*VECA + SN*VECB) (-SN*VECA + CS*VECB)]

where:

- CS**(2) + SN**(2) = 1,
- -SN*VECA(1) + CS*VECB(1) = 0,
- CS*VECA(1) + SN*VECB(1) >= 0.

and ROT is a 2-by-2 matrix defined by

(+CS -SN)

(+SN +CS)

Arguments

VECA (INPUT/OUTPUT) real(stnd), dimension(:) The first vector to be rotated.

On output, CS*VECA + SN*VECB overwrites VECA.

VECB (INPUT/OUTPUT) real(stnd), dimension(:) The second vector to be rotated.

On output, -SN*VECA + CS*VECB overwrites VECB.

CS (OUTPUT) real(stnd) The cosine of the rotation.

SN (OUTPUT) real(stnd) The sine of the rotation.

Further Details

It is assumed that VECA and VECB have the same size.

6.6.12 subroutine givens_mat_left (mat)

Purpose

GIVENS_MAT_LEFT transforms the matrix MAT to upper trapezoidal form by applying a serie of Givens plane rotations on the rows of MAT.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) The matrix to be transformed.

On output, the transformed matrix overwrites MAT.

6.6.13 subroutine givens_mat_right (mat)

Purpose

GIVENS_MAT_RIGHT transforms the matrix MAT to lower trapezoidal form by applying a serie of Givens plane rotations on the columns of MAT.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) The matrix to be transformed.

On output, the transformed matrix overwrites MAT.

6.6.14 subroutine givens_vec_mat_left (vec, mat)

Purpose

GIVENS_VEC_MAT_LEFT defines and applies a serie of Givens rotations on a n-vector VEC and on the rows of a p-by-n matrix MAT. The rotations are designed to annilhate all the elements of the first column of MAT.

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On input, the n-vector to rotate. VEC(1) is used to define the rotations.

On output, the transformed vector overwrites VEC.

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On input, the matrix to be transformed. MAT(:,1) is used to define the rotations.

On output, the transformed matrix overwrites MAT and MAT(:,1) is equal to zero.

Further Details

It is assumed that size(VEC) = size(MAT, 2) .

6.6.15 subroutine givens_vec_mat_right (vec, mat)

Purpose

GIVENS_VEC_MAT_RIGHT defines and applies a serie of Givens rotations on a n-vector VEC and on the columns of a n-by-p matrix MAT. The rotations are designed to annilhate all the elements of the first row of MAT.

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On input, the n-vector to rotate. VEC(1) is used to define the rotations.

On output, the transformed vector overwrites VEC.

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On input, the matrix to be transformed. MAT(1,:) is used to define the rotations.

On output, the transformed matrix overwrites MAT and MAT(1,:) is equal to zero.

Further Details

It is assumed that size(VEC) = size(MAT, 1) .

6.6.16 subroutine define_rot_fastgivens (x1, x2, d1, d2, beta, alpha, type_rot)

Purpose

DEFINE_ROT_FASTGIVENS generates a fast Givens plane rotation H (defined by BETA, ALPHA, and TYPE_ROT on output) and updated scale factors (D1 and D2), which zero X2. That is,

(X1 X2) H = (R 0)

, where H is equal to

- (1 0) ,if TYPE_ROT = 0.

  (0 1)

- (1 0) (1 A) ,if TYPE_ROT = 1.

  (B 1) (0 1)

- (1 A) (1 0) ,if TYPE_ROT = 2.

  (0 1) (B 1)

- (0 -1) (1 0) ,if TYPE_ROT = 3.

  (1 A) (-B 1)

- (B 1) (1 A) ,if TYPE_ROT = 4.

  (1 0) (0 -1)

Furtermore, if on input, Y1 = X1*SQRT(D1) and Y2 = X2*SQRT(D2), then on output

(X1 X2) H diag(SQRT(D1) SQRT(D2)) = ((X1*H11 + X2*H21)*SQRT(D1) 0)

is equal to

(Y1 Y2) ROT = ((Y1*CS + Y2*SN) 0)

where CS and SN define a standard Givens plane rotation, ROT, which zeros Y2. Thus, ROT is equal to

(+CS -SN)

(+SN +CS)

with CS**(2) + SN**(2) = 1.

Arguments

X1 (INPUT) real(stnd) First component of vector to be transformed.

X2 (INPUT) real(stnd) Second component of vector to be transformed.

D1 (INPUT/OUTPUT) real(stnd) On input, first scale factor.

On output, D1 is replaced with the update scale factor.

D2 (INPUT/OUTPUT) real(stnd) On input, second scale factor.

On output, D2 is replaced with the update scale factor.

BETA (OUTPUT) real(stnd) The real scalar B which defines the transformation matrix H.

ALPHA (OUTPUT) real(stnd) The real scalar A which defines the transformation matrix H.

TYPE_ROT (OUTPUT) integer(i2b) Integer which defines the transformation matrix H.

Further Details

X1 and X2 are unchanged on return.

It is assumed that D1 and D2 are positive scalars.

IF D1>=D2, D1 is diminished and D2 is augmented. IF D1<D2, D2 is diminished and D1 is augmented. The decrease or increase in magnitude of D1 and D2 are bounded by 1/2 and 2, respectively.

Normally, the subprogram APPLY_ROT_FASTGIVENS(VECX1, VECX2, BETA, ALPHA, TYPE_ROT) will next be called to apply the rotation to a n-by-2 matrix [VECX1 VECX2].

This subroutine is a square root free implementation of the two-way branch algorithm (fast plane rotations with dynamic scaling to avoid overflow/underflow) described in reference (1).

For further details, see:

(1) **Anda, A.A. and Park, H., 1994:** Fast plane rotations with dynamic scaling. Siam J. Matrix Anal. Appl., 15, 162-174.

6.6.17 subroutine define_rot_fastgivens2 (x1, x2, d1, d2, beta, alpha, type_rot)

Purpose

DEFINE_ROT_FASTGIVENS2 generates a fast Givens plane rotation H (defined by BETA, ALPHA, and TYPE_ROT on output) and updated scale factors (D1 and D2), which zero X2. That is,

(X1 X2) H = (R 0)

, where H is equal to

- (1 0) ,if TYPE_ROT = 0.

  (0 1)

- (1 0) (1 A) ,if TYPE_ROT = 1.

  (B 1) (0 1)

- (1 A) (1 0) ,if TYPE_ROT = 2.

  (0 1) (B 1)

- (0 -1) (1 0) ,if TYPE_ROT = 3.

  (1 A) (-B 1)

- (B 1) (1 A) ,if TYPE_ROT = 4.

  (1 0) (0 -1)

Furtermore, if on input, Y1 = X1*D1 and Y2 = X2*D2, then on output

(X1 X2) H diag(D1 D2) = ((X1*H11 + X2*H21)*D1 0)

is equal to

(Y1 Y2) ROT = ((Y1*CS + Y2*SN) 0)

where CS and SN define a standard Givens plane rotation, ROT, which zeros Y2. Thus, ROT is equal to

(+CS -SN)

(+SN +CS)

with CS**(2) + SN**(2) = 1.

Arguments

X1 (INPUT) real(stnd) First component of vector to be transformed.

X2 (INPUT) real(stnd) Second component of vector to be transformed.

D1 (INPUT/OUTPUT) real(stnd) On input, first scale factor.

On output, D1 is replaced with the update scale factor.

D2 (INPUT/OUTPUT) real(stnd) On input, second scale factor.

On output, D2 is replaced with the update scale factor.

BETA (OUTPUT) real(stnd) The real scalar B which defines the transformation matrix H.

ALPHA (OUTPUT) real(stnd) The real scalar A which defines the transformation matrix H.

TYPE_ROT (OUTPUT) integer(i2b) Integer which defines the transformation matrix H.

Further Details

X1 and X2 are unchanged on return.

It is assumed that D1 and D2 are positive scalars.

IF D1>=D2, D1 is diminished and D2 is augmented. IF D1<D2, D2 is diminished and D1 is augmented. The decrease or increase in magnitude of D1 and D2 are bounded by 1/sqrt(2) and sqrt(2), respectively.

Normally, the subprogram APPLY_ROT_FASTGIVENS(VECX1, VECX2, BETA, ALPHA, TYPE_ROT) will next be called to apply the rotation to a n-by-2 matrix [VECX1 VECX2].

This subroutine is an implementation of the two-way branch algorithm (fast plane rotations with dynamic scaling to avoid overflow/underflow) described in reference (1).

For further details, see:

(1) **Anda, A.A. and Park, H., 1994:** Fast plane rotations with dynamic scaling. Siam J. Matrix Anal. Appl., 15, 162-174.

6.6.18 subroutine apply_rot_fastgivens (y1, y2, beta, alpha, type_rot)

Purpose

APPLY_ROT_FASTGIVENS applies a fast Givens plane rotation H (defined by BETA, ALPHA, and TYPE_ROT on input) to the vector (Y1 Y2). That is,

(Y1 Y2) H = ((Y1*H11 + Y2*H21) (Y1*H12 + Y2*H22))

where H is a 2-by-2 matrix defined as

(H11 H12)

(H21 H22)

More precisely, H takes one of the following forms:

- (1 0) ,if TYPE_ROT = 0.

  (0 1)

- (1 0) (1 A) ,if TYPE_ROT = 1.

  (B 1) (0 1)

- (1 A) (1 0) ,if TYPE_ROT = 2.

  (0 1) (B 1)

- (0 -1) (1 0) ,if TYPE_ROT = 3.

  (1 A) (-B 1)

- (B 1) (1 A) ,if TYPE_ROT = 4.

  (1 0) (0 -1)

Arguments

Y1 (INPUT/OUTPUT) real(stnd) The first component of vector to be transformed.

On output, Y1*H11 + Y2*H21 overwrites Y1.

Y2 (INPUT/OUTPUT) real(stnd) The second component of vector to be transformed.

On output, Y1*H12 + Y2*H22 overwrites Y2.

BETA (OUTPUT) real(stnd) The real scalar B which defines the transformation matrix H.

ALPHA (OUTPUT) real(stnd) The real scalar A which defines the transformation matrix H.

TYPE_ROT (INPUT) integer(i2b) Integer which defines the transformation matrix H.

Further Details

Normally, the subprogram APPLY_ROT_FASTGIVENS(Y1, Y2, BETA, ALPHA, TYPE_ROT) will be called to apply the transformation to the vector (Y1 Y2) after a call to DEFINE_ROT_FASTGIVENS(X1, X2, BETA, ALPHA, TYPE_ROT) or DEFINE_ROT_FASTGIVENS2(X1, X2, BETA, ALPHA, TYPE_ROT).

6.6.19 subroutine apply_rot_fastgivens (vecy1, vecy2, beta, alpha, type_rot)

Purpose

APPLY_ROT_FASTGIVENS applies a fast Givens plane rotation H (defined by BETA, ALPHA, and TYPE_ROT on input) to the n-by-2 matrix [VECY1 VECY2]. That is,

[VECY1 VECY2] H = [(VECY1*H11 + VECY2*H21) (VECY1*H12 + VECY2*H22)]

where H is a 2-by-2 matrix defined as

(H11 H12)

(H21 H22)

More precisely, H takes one of the following forms:

- (1 0) ,if TYPE_ROT = 0.

  (0 1)

- (1 0) (1 A) ,if TYPE_ROT = 1.

  (B 1) (0 1)

- (1 A) (1 0) ,if TYPE_ROT = 2.

  (0 1) (B 1)

- (0 -1) (1 0) ,if TYPE_ROT = 3.

  (1 A) (-B 1)

- (B 1) (1 A) ,if TYPE_ROT = 4.

  (1 0) (0 -1)

Arguments

VECY1 (INPUT/OUTPUT) real(stnd), dimension(:) The first vector to be transformed.

On output, VECY1*H11 + VECY2*H21 overwrites VECY1.

VECY2 (INPUT/OUTPUT) real(stnd), dimension(:) The second vector to be transformed.

On output, VECY1*H12 + VECY2*H22 overwrites VECY2.

BETA (OUTPUT) real(stnd) The real scalar B which defines the transformation matrix H.

ALPHA (OUTPUT) real(stnd) The real scalar A which defines the transformation matrix H.

TYPE_ROT (INPUT) integer(i2b) Integer which defines the transformation matrix H.

Further Details

Normally, the subprogram APPLY_ROT_FASTGIVENS(VECY1, VECY2, BETA, ALPHA, TYPE_ROT) will be called to apply the transformation to the n-by-2 matrix [VECY1 VECY2] after a call to DEFINE_ROT_FASTGIVENS(VECY1(1), VECY2(1), BETA, ALPHA, TYPE_ROT) or DEFINE_ROT_FASTGIVENS2(VECY1(1), VECY2(1), BETA, ALPHA, TYPE_ROT).

It is assumed that VECY1 and VECY2 have the same size.

6.6.20 subroutine fastgivens_vec (vecx1, vecx2, d1, d2)

Purpose

FASTGIVENS generates and applies a fast Givens plane rotation H to the n-by-2 matrix [VECX1 VECX2]. The rotation is designed to zero VECX2(1). That is,

[VECX1 VECX2] H = [(VECX1*H11 + VECX2*H21) (VECX1*H12 + VECX2*H22)]

, where VECX1(1)*H12 + VECX2(1)*H22 = 0 and H is the 2-by-2 matrix:

(H11 H12)

(H21 H22)

Furthermore, the scale factors (D1 and D2) are updated accordingly. That is, if on input:

[Y1 Y2] = [VECX1 VECX2] diag(SQRT(D1) SQRT(D2))

then on output:

[VECX1 VECX2] diag(SQRT(D1) SQRT(D2)) = [Y1 Y2] ROT = [(Y1*CS + Y2*SN) (-SN*Y1 + CS*Y2)]

where CS and SN define a standard Givens 2-by-2 plane rotation, ROT, which zeros -SN*Y1(1) + CS*Y2(1). Thus, ROT has the following structure:

(+CS -SN)

(+SN +CS)

with CS**(2) + SN**(2) = 1.

Arguments

VECX1 (INPUT/OUTPUT) real(stnd), dimension(:) The first vector to be transformed.

On output, VECX1*H11 + VECX2*H21 overwrites VECX1.

VECX2 (INPUT/OUTPUT) real(stnd), dimension(:) The second vector to be transformed.

On output, VECX1*H12 + VECX2*H22 overwrites VECX2.

D1 (INPUT/OUTPUT) real(stnd) On input, first scale factor.

On output, D1 is replaced with the update scale factor.

D2 (INPUT/OUTPUT) real(stnd) On input, second scale factor.

On output, D2 is replaced with the update scale factor.

Further Details

It is assumed that D1 and D2 are positive scalars and that VECX1 and VECX2 have the same size.

IF D1>=D2, D1 is diminished and D2 is augmented. IF D1<D2, D2 is diminished and D1 is augmented. The decrease or increase in magnitude of D1 and D2 are bounded by 1/2 and 2, respectively.

This subroutine is a square root free implementation of the two-way branch algorithm (e.g. fast plane rotations with dynamic scaling to avoid overflow/underflow) described in reference (1).

For further details, see:

(1) **Anda, A.A. and Park, H., 1994:** Fast plane rotations with dynamic scaling. Siam J. Matrix Anal. Appl., 15, 162-174.

6.6.21 subroutine fastgivens_vec (vecx1, vecx2, d1, d2, beta, alpha, type_rot)

Purpose

FASTGIVENS generates and applies a fast Givens plane rotation H (defined by BETA, ALPHA and TYPE_ROT on output) to the n-by-2 matrix [VECX1 VECX2]. The rotation is designed to zero VECX2(1). That is,

[VECX1 VECX2] H = [(VECX1*H11 + VECX2*H21) (VECX1*H12 + VECX2*H22)]

, where VECX1(1)*H12 + VECX2(1)*H22 = 0 and H is the 2-by-2 matrix:

(H11 H12)

(H21 H22)

and H takes one of the following forms:

- (1 0) ,if TYPE_ROT = 0.

  (0 1)

- (1 0) (1 A) ,if TYPE_ROT = 1.

  (B 1) (0 1)

- (1 A) (1 0) ,if TYPE_ROT = 2.

  (0 1) (B 1)

- (0 -1) (1 0) ,if TYPE_ROT = 3.

  (1 A) (-B 1)

- (B 1) (1 A) ,if TYPE_ROT = 4.

  (1 0) (0 -1)

Furthermore, the scale factors (D1 and D2) are updated accordingly. That is, if on input:

[Y1 Y2] = [VECX1 VECX2] diag(SQRT(D1) SQRT(D2))

then on output:

[VECX1 VECX2] diag(SQRT(D1) SQRT(D2)) = [Y1 Y2] ROT = [(Y1*CS + Y2*SN) (-SN*Y1 + CS*Y2)]

where CS and SN define a standard Givens 2-by-2 plane rotation, ROT, which zeros -SN*Y1(1) + CS*Y2(1). Thus, ROT has the following structure:

(+CS -SN)

(+SN +CS)

with CS**(2) + SN**(2) = 1.

Arguments

VECX1 (INPUT/OUTPUT) real(stnd), dimension(:) The first vector to be transformed.

On output, VECX1*H11 + VECX2*H21 overwrites VECX1.

VECX2 (INPUT/OUTPUT) real(stnd), dimension(:) The second vector to be transformed.

On output, VECX1*H12 + VECX2*H22 overwrites VECX2.

D1 (INPUT/OUTPUT) real(stnd) On input, first scale factor.

On output, D1 is replaced with the update scale factor.

D2 (INPUT/OUTPUT) real(stnd) On input, second scale factor.

On output, D2 is replaced with the update scale factor.

BETA (OUTPUT) real(stnd) The real scalar B which defines the transformation matrix H.

ALPHA (OUTPUT) real(stnd) The real scalar A which defines the transformation matrix H.

TYPE_ROT (OUTPUT) integer(i2b) Integer which defines the transformation matrix H.

Further Details

It is assumed that D1 and D2 are positive scalars and that VECX1 and VECX2 have the same size.

IF D1>=D2, D1 is diminished and D2 is augmented. IF D1<D2, D2 is diminished and D1 is augmented. The decrease or increase in magnitude of D1 and D2 are bounded by 1/2 and 2, respectively.

This subroutine is a square root free implementation of the two-way branch algorithm (e.g. fast plane rotations with dynamic scaling to avoid overflow/underflow) described in reference (1).

For further details, see:

(1) **Anda, A.A. and Park, H., 1994:** Fast plane rotations with dynamic scaling. Siam J. Matrix Anal. Appl., 15, 162-174.

6.6.22 `subroutine fastgivens2_vec ( vecx1, vecx2, d1, d2 )`

Purpose

FASTGIVENS2 generates and applies a fast Givens plane rotation H to the n-by-2 matrix [VECX1 VECX2]. The rotation is designed to zero VECX2(1). That is,

[VECX1 VECX2] H = [(VECX1*H11 + VECX2*H21) (VECX1*H12 + VECX2*H22)]

, where VECX1(1)*H12 + VECX2(1)*H22 = 0 and H is the 2-by-2 matrix:

(H11 H12)

(H21 H22)

, where VECX1(1)*H12 + VECX2(1)*H22 = 0.

Furthermore, the scale factors (D1 and D2) are updated accordingly. That is, if on input:

[Y1 Y2] = [VECX1 VECX2] diag(D1 D2)

then on output:

[VECX1 VECX2] diag(D1 D2) = [Y1 Y2] ROT = [(Y1*CS + Y2*SN) (-SN*Y1 + CS*Y2)]

where CS and SN define a standard Givens 2-by-2 plane rotation, ROT, which zeros -SN*Y1(1) + CS*Y2(1). Thus, ROT has the following structure:

(+CS -SN)

(+SN +CS)

with CS**(2) + SN**(2) = 1.

Arguments

VECX1 (INPUT/OUTPUT) real(stnd), dimension(:) The first vector to be transformed.

On output, VECX1*H11 + VECX2*H21 overwrites VECX1.

VECX2 (INPUT/OUTPUT) real(stnd), dimension(:) The second vector to be transformed.

On output, VECX1*H12 + VECX2*H22 overwrites VECX2.

D1 (INPUT/OUTPUT) real(stnd) On input, first scale factor.

On output, D1 is replaced with the update scale factor.

D2 (INPUT/OUTPUT) real(stnd) On input, second scale factor.

On output, D2 is replaced with the update scale factor.

Further Details

It is assumed that D1 and D2 are positive scalars and that VECX1 and VECX2 have the same size.

IF D1>=D2, D1 is diminished and D2 is augmented. IF D1<D2, D2 is diminished and D1 is augmented. The decrease or increase in magnitude of D1 and D2 are bounded by 1/sqrt(2) and sqrt(2), respectively.

This subroutine is an implementation of the two-way branch algorithm (e.g. fast plane rotations with dynamic scaling to avoid overflow/underflow) described in reference (1).

For further details, see:

(1) **Anda, A.A. and Park, H., 1994:** Fast plane rotations with dynamic scaling. Siam J. Matrix Anal. Appl., 15, 162-174.

6.6.23 `subroutine fastgivens2_vec ( vecx1, vecx2, d1, d2, beta, alpha, type_rot )`

Purpose

FASTGIVENS2 generates and applies a fast Givens plane rotation H (defined by BETA, ALPHA and TYPE_ROT on output) to the n-by-2 matrix [VECX1 VECX2]. The rotation is designed to zero VECX2(1). That is,

[VECX1 VECX2] H = [(VECX1*H11 + VECX2*H21) (VECX1*H12 + VECX2*H22)]

, where VECX1(1)*H12 + VECX2(1)*H22 = 0 and H is the 2-by-2 matrix:

(H11 H12)

(H21 H22)

and takes one of the following forms:

- (1 0) ,if TYPE_ROT = 0.

  (0 1)

- (1 0) (1 A) ,if TYPE_ROT = 1.

  (B 1) (0 1)

- (1 A) (1 0) ,if TYPE_ROT = 2.

  (0 1) (B 1)

- (0 -1) (1 0) ,if TYPE_ROT = 3.

  (1 A) (-B 1)

- (B 1) (1 A) ,if TYPE_ROT = 4.

  (1 0) (0 -1)

Furthermore, the scale factors (D1 and D2) are updated accordingly. That is, if on input:

[Y1 Y2] = [VECX1 VECX2] diag(D1 D2)

then on output:

[VECX1 VECX2] diag(D1 D2) = [Y1 Y2] ROT = [(Y1*CS + Y2*SN) (-SN*Y1 + CS*Y2)]

where CS and SN define a standard Givens 2-by-2 plane rotation, ROT, which zeros -SN*Y1(1) + CS*Y2(1). Thus, ROT has the following structure:

(+CS -SN)

(+SN +CS)

with CS**(2) + SN**(2) = 1.

Arguments

VECX1 (INPUT/OUTPUT) real(stnd), dimension(:) The first vector to be transformed.

On output, VECX1*H11 + VECX2*H21 overwrites VECX1.

VECX2 (INPUT/OUTPUT) real(stnd), dimension(:) The second vector to be transformed.

On output, VECX1*H12 + VECX2*H22 overwrites VECX2.

D1 (INPUT/OUTPUT) real(stnd) On input, first scale factor.

On output, D1 is replaced with the update scale factor.

D2 (INPUT/OUTPUT) real(stnd) On input, second scale factor.

On output, D2 is replaced with the update scale factor.

BETA (OUTPUT) real(stnd) The real scalar B which defines the transformation matrix H.

ALPHA (OUTPUT) real(stnd) The real scalar A which defines the transformation matrix H.

TYPE_ROT (OUTPUT) integer(i2b) Integer which defines the transformation matrix H.

Further Details

It is assumed that D1 and D2 are positive scalars and that VECX1 and VECX2 have the same size.

IF D1>=D2, D1 is diminished and D2 is augmented. IF D1<D2, D2 is diminished and D1 is augmented. The decrease or increase in magnitude of D1 and D2 are bounded by 1/sqrt(2) and sqrt(2), respectively.

This subroutine is an implementation of the two-way branch algorithm (e.g. fast plane rotations with dynamic scaling to avoid overflow/underflow) described in reference (1).

For further details, see:

(1) **Anda, A.A. and Park, H., 1994:** Fast plane rotations with dynamic scaling. Siam J. Matrix Anal. Appl., 15, 162-174.

6.6.24 `subroutine fastgivens_mat_left ( mat, matd )`

Purpose

FASTGIVENS_MAT_LEFT reduces the matrix MAT to upper trapezoidal form by applying a serie of fast Givens plane rotations on the rows of MAT.

The (row) scale factors (MATD) are updated accordingly.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) The matrix to be transformed.

On output, the transformed matrix overwrites MAT.

MATD (INPUT/OUTPUT) real(stnd), dimension(:) On input, scale factors associated with the rows of MAT.

On output, MATD is replaced with the update scale factors.

Further Details

It is assumed that size(MATD) = size(MAT, 1) and that MATD is a positive vector.

See description of FASTGIVENS for further details.

6.6.25 `subroutine fastgivens_mat_right ( mat, matd )`

Purpose

FASTGIVENS_MAT_RIGHT reduces the matrix MAT to lower trapezoidal form by applying a serie of fast Givens plane rotations on the columns of MAT.

The (column) scale factors (MATD) are updated accordingly.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) The matrix to be transformed.

On output, the transformed matrix overwrites MAT.

MATD (INPUT/OUTPUT) real(stnd), dimension(:) On input, scale factors associated with the columns of MAT.

On output, MATD is replaced with the update scale factors.

Further Details

It is assumed that size(MATD) = size(MAT, 2) and that MATD is a positive vector.

See description of FASTGIVENS for further details.

6.6.26 `subroutine fastgivens_vec_mat_left ( vec, mat, vecd, matd )`

Purpose

FASTGIVENS_VEC_MAT_LEFT defines and applies a serie of fast Givens plane rotations on the n-vector VEC and on the rows of a m-by-n matrix MAT. The rotations are designed to annilhate all the elements of the first column of MAT.

The (row) scale factors (VECD and MATD) are updated accordingly.

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On input, the n-vector to rotate. VEC(1) is used to define the rotations.

On output, the transformed vector overwrites VEC.

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On input, the matrix to be transformed. MAT(:,1) is used to define the rotations.

On output, the transformed matrix overwrites MAT and MAT(:,1) is equal to zero (within numerical accuracy).

VECD (INPUT/OUTPUT) real(stnd) On input, scale factor associated with the n-vector VEC.

On output, VECD is replaced with the update scale factor.

MATD (INPUT/OUTPUT) real(stnd), dimension(:) On input, scale factors associated with the rows of MAT.

On output, MATD is replaced with the update scale factors.

Further Details

It is assumed that:

- size(VEC) = size(MAT, 2);
- VECD is a positive scalar;
- size(MATD) = size(MAT,1) and that MATD is a positive vector.

See description of FASTGIVENS for further details.

6.6.27 subroutine fastgivens_vec_mat_right (vec, mat, vecd, matd)

Purpose

FASTGIVENS_VEC_MAT_RIGHT defines and applies a serie of fast Givens plane rotations on the m-vector VEC and on the columns of a m-by-n matrix MAT. The rotations are designed to annilhate all the elements of the first row of MAT.

The (column) scale factors (VECD and MATD) are updated accordingly.

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On input, the m-vector to rotate. VEC(1) is used to define the rotations.

On output, the transformed vector overwrites VEC.

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On input, the matrix to be transformed. MAT(1,:) is used to define the rotations.

On output, the transformed matrix overwrites MAT and MAT(1,:) is equal to zero (within numerical accuracy).

VECD (INPUT/OUTPUT) real(stnd) On input, scale factor associated with the m-vector VEC.

On output, VECD is replaced with the update scale factor.

MATD (INPUT/OUTPUT) real(stnd), dimension(:) On input, scale factors associated with the columns of MAT.

On output, MATD is replaced with the update scale factors.

Further Details

It is assumed that:

- size(VEC) = size(MAT, 1);
- VECD is a positive scalar;
- size(MATD) = size (MAT, 2) and that MATD is a positive vector.

See description of FASTGIVENS for further details.

6.7 Module_Hous_Procedures

Copyright 2024 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING HOUSEHOLDER REFLECTIONS.

LATEST REVISION : 11/03/2024

6.7.1 `subroutine hous1 ( u, tau )`

Purpose

HOUS1 generates a real elementary reflector H of order n, such that

H * X = D , with H' * H = I and D' = (beta 0)

where beta is scalar and X is an n-element real vector.

H is represented in the form

H = I + tau * (v * v') ,

where tau is a real scalar and v is an n-element real vector with v(1) = 1.

If the elements of X(2:n) are all zero or size(X)=1, then tau = 0 and H is taken to be the unit matrix.

Otherwise 1 <= tau <= 2.

Arguments

U (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the vector X.

On exit, it is overwritten with the vector [beta v(2:n)].

TAU (OUTPUT) real(stnd) On exit, the value tau.

Further Details

This subroutine is based on the routine DLARFG in LAPACK77 (version 3) with improvements suggested in references (1), (2) and (3).

For more details, see:

(1) **Anderson, E., and Fahey, M., 1997:** Performance improvements to LAPACK for the Cray Scientific Library. LAPACK Working Note No 126.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.7.2 subroutine hous1 (u, tau, beta)

Purpose

HOUS1 generates a real elementary reflector H of order n, such that

H * X = D , with H' * H = I and D' = (beta 0)

where beta is scalar and X is an n-element real vector.

H is represented in the form

H = I + tau * (v * v') ,

where tau is a real scalar and v is an n-element real vector with v(1) = 1.

If the elements of X(2:n) are all zero or size(X)=1, then tau = 0 and H is taken to be the unit matrix.

Otherwise 1 <= tau <= 2.

Arguments

U (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the vector x.

On exit, it is overwritten with the vector [1 v(2:n)].

TAU (OUTPUT) real(stnd) On exit, the value tau.

BETA (OUTPUT) real(stnd) On exit, the value beta.

Further Details

This subroutine is based on the routine DLARFG in LAPACK77 (version 3) with improvements suggested in references (1), (2) and (3).

For more details, see:

(1) **Anderson, E., and Fahey, M., 1997:** Performance improvements to LAPACK for the Cray Scientific Library. LAPACK Working Note No 126.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.7.3 subroutine hous2 (pivot, u, tau)

Purpose

HOUS2 generates a real elementary reflector H of order n, such that

H * X = D , with H' * H = I , X' = (alpha x) and D' = (beta 0)

where alpha and beta are scalars, and x is an (n-1)-element real vector.

H is represented in the form

H = I + tau * (v * v') ,

where tau is a real scalar and v is an n-element real vector with v(1) = 1.

If the elements of x are all zero, then tau = 0 and H is taken to be the unit matrix.

Otherwise 1 <= tau <= 2.

Arguments

PIVOT (INPUT/OUTPUT) real(stnd) On entry, the value alpha. On exit, it is overwritten with the value beta.

U (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the (n-1)-element vector x.

On exit, it is overwritten with the vector v(2:n) .

TAU (OUTPUT) real(stnd) On exit, the value tau.

Further Details

This subroutine is based on the routine DLARFG in LAPACK77 (version 3) with improvements suggested in references (1), (2) and (3).

For more details, see:

(1) **Anderson, E., and Fahey, M., 1997:** Performance improvements to LAPACK for the Cray Scientific Library. LAPACK Working Note No 126.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.7.4 `subroutine apply_hous1 ( u, tau, vec )`

Purpose

APPLY_HOUS1 applies a real elementary reflector H generated by HOUS1 to a real vector C. H is represented in the form

H = I + tau * (v * v') ,

where tau is a real scalar and v is an n-element real vector with v(1) = 1.

If tau = 0, then H is taken to be the unit matrix.

Arguments

U (INPUT/OUTPUT) real(stnd), dimension(:) On entry, U(2:) contains the vector v(2:) in the representation of H as output by HOUS1. U is not used if tau = 0.

U is restored on exit.

TAU (INPUT) real(stnd) The value tau in the representation of H as output by HOUS1.

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the real vector C.

On exit, C is overwritten by the vector H * C

Further Details

It is assumed that size(VEC)>=size(U) .

6.7.5 subroutine apply_hous1 (u, tau, mat, left)

Purpose

APPLY_HOUS1 applies a real elementary reflector H generated by HOUS1 to a real n-by-m or m-by-n matrix, C, from the left or the right. H is represented in the form

H = I + tau * (v * v') ,

where tau is a real scalar and v is an n-element real vector with v(1) = 1.

If tau = 0, then H is taken to be the unit matrix.

Arguments

U (INPUT/OUTPUT) real(stnd), dimension(:) On entry, U(2:) contains the vector v(2:) in the representation of H as output by HOUS1. U is not used if tau = 0.

U is restored on exit.

TAU (INPUT) real(stnd) The value tau in the representation of H as output by HOUS1.

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the n-by-m or m-by-n matrix C.

On exit, C is overwritten by the matrix H * C (LEFT=true) or C * H (LEFT=false).

LEFT (INPUT) logical(lgl) If:

- LEFT=true, H is applied to the real matrix C from the left;
- LEFT=false, H is applied to the real matrix C from the right.

Further Details

It is assumed that:

- size(MAT, 1) >= size(U) if LEFT=true;
- size(MAT, 2) >= size(U) if LEFT=false.

6.7.6 subroutine apply_hous2 (u, tau, piv, vec)

Purpose

APPLY_HOUS2 applies a real n-by-n elementary reflector H generated by HOUS2 to a real vector C. H is represented in the form

H = I + tau * (v * v') ,

where tau is a real scalar and v is an n-element real vector with v(1) = 1.

If tau = 0, then H is taken to be the unit matrix.

Arguments

U (INPUT) real(stnd), dimension(:) On entry, U contains the vector v(2:n) in the representation of H as output by HOUS2. U is not used if tau = 0.

TAU (INPUT) real(stnd) The value tau in the representation of H as output by HOUS2.

PIV (INPUT/OUTPUT) real(stnd) On entry, the scalar C[1].

On exit, PIV is overwritten by the scalar (H * C)[1].

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the real vector C[2:].

On exit, C is overwritten by the vector (H * C)[2:].

Further Details

It is assumed that size(VEC) >= size(U).

6.7.7 subroutine apply_hous2 (u, tau, vec_piv, mat, left)

Purpose

APPLY_HOUS2 applies a real n-by-n elementary reflector H generated by HOUS2 to a real n-by-m or m-by-n matrix, C, from the left or the right. H is represented in the form

H = I + tau * (v * v') ,

where tau is a real scalar and v is an n-element real vector with v(1) = 1.

If tau = 0, then H is taken to be the unit matrix.

Arguments

U (INPUT) real(stnd), dimension(:) On entry, U contains the vector v(2:n) in the representation of H as output by HOUS2. U is not used if tau = 0.

TAU (INPUT) real(stnd) The value tau in the representation of H as output by HOUS2.

VEC_PIV (INPUT/OUTPUT) real(stnd), dimension(:) If LEFT=true:

- On entry, the row_vector C[1,:];
- On exit, VEC_PIV is overwritten by the vector (H * C)[1,:].

If LEFT=false:

- On entry, the column_vector C[:,1];
- On exit, VEC_PIV is overwritten by the vector (C * H)[:,1].

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) If LEFT=true:

- On entry, the (n-1)-by-m matrix C[2:,:];
- On exit, MAT is overwritten by the matrix (H * C)[2:,:].

If LEFT=false:

- On entry, the m-by-(n-1) matrix C[:,2:];
- On exit, MAT is overwritten by the matrix (C * H)[:,2:].

LEFT (INPUT) logical(lgl) If:

- LEFT=true, H is applied to the real matrix C from the left;
- LEFT=false, H is applied to the real matrix C from the right.

Further Details

It is assumed that:

- size(MAT, 1) >= size(U) and size(MAT, 2) >= size(VEC_PIV) if LEFT=true;
- size(MAT, 2) >= size(U) and size(MAT, 1) >= size(VEC_PIV) if LEFT=false.

6.7.8 `subroutine h1 ( u, beta, tau )`

Purpose

H1 generates a real elementary reflector H of order n, such that

H * X = D , with H' * H = I and D' = (beta 0)

where beta is scalar and X is an n-element real vector.

H is represented in the form

H = I + tau * (v * v') ,

where tau is a real scalar and v is a real n-element vector.

Arguments

U (INPUT/OUTPUT) real(stnd), dimension(:) On entry, U contains the pivot vector X.

On exit, U contains the vector v of the Householder reflector.

BETA (OUTPUT) real(stnd) On exit, the value beta.

TAU (OUTPUT) real(stnd) On exit, the value tau.

Further Details

On output, H is the identity matrix if TAU = 0.

This subroutine is adapted from:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Hanson, R.J., and Hopkins, T., 2017:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24 (December 2017), 23 pages.

6.7.9 subroutine h1 (u, beta, tau, vec)

Purpose

H1 generates a real elementary reflector H of order n, such that

H * X = D , with H' * H = I and D' = (beta 0)

where beta is scalar and X is an n-element real vector.

H is represented in the form

H = I + tau * (v * v') ,

where tau is a real scalar and v is a real n-element vector.

The real elementary reflector H is then applied to a real vector C .

Arguments

U (INPUT/OUTPUT) real(stnd), dimension(:) On entry, U contains the pivot vector X.

On exit, U contains the vector v of the Householder reflector.

BETA (OUTPUT) real(stnd) On exit, the value beta.

TAU (OUTPUT) real(stnd) On exit, the value tau.

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the real vector C .

On exit, C is overwritten by the vector H * C .

Further Details

On output, H is the identity matrix if TAU = 0.

It is assumed that size(VEC) >= size(U) = n .

This subroutine is adapted from:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Hanson, R.J., and Hopkins, T., 2017:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24 (December 2017), 23 pages.

6.7.10 subroutine h1 (u, beta, tau, mat, left)

Purpose

H1 generates a real elementary reflector H of order n, such that

H * X = D , with H' * H = I and D' = (beta 0)

where beta is scalar and X is an n-element real vector.

H is represented in the form

H = I + tau * (v * v') ,

where tau is a real scalar and v is a real n-element vector.

The real elementary reflector H is then applied to a real matrix C from the left or the right.

Arguments

U (INPUT/OUTPUT) real(stnd), dimension(:) On entry, U contains the pivot vector X.

On exit, U contains the vector v of the Householder reflector.

BETA (OUTPUT) real(stnd) On exit, the value beta.

TAU (OUTPUT) real(stnd) On exit, the value tau.

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the n-by-m or m-by-n real matrix C.

On exit, C is overwritten by the matrix H * C (if LEFT=true) or C * H (if LEFT=false).

LEFT (INPUT) logical(lgl) If:

- LEFT=true, H is applied to the real matrix C from the left;
- LEFT=false, H is applied to the real matrix C from the right.

Further Details

On output, H is the identity matrix if TAU = 0.

It is assumed that:

- size(MAT, 1) >= size(U) if LEFT=true;
- size(MAT, 2) >= size(U) if LEFT=false.

This subroutine is adapted from:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Hanson, R.J., and Hopkins, T., 2017:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24 (December 2017), 23 pages.

6.7.11 `subroutine h2 ( beta, u, up, tau )`

Purpose

H2 generates a real elementary reflector H of order n, such that

H * X = D , with H' * H = I , X' = (alpha x) and D' = (beta 0)

where alpha and beta are scalars, and x is an (n-1)-element real vector.

H is represented in the form

H = I + tau * (v * v') ,

where tau is a real scalar and v is a real n-element vector.

Arguments

BETA (INPUT/OUTPUT) real(stnd) On entry, the value alpha.

On exit, it is overwritten with the value beta.

U (INPUT/OUTPUT) real(stnd), dimension(:) On entry, U contains the pivot (n-1)-element vector x.

On exit, U contains the vector v(2:) of the Householder reflector.

UP (OUTPUT) real(stnd) On exit, UP contains the value v(1) of the Householder reflector.

TAU (OUTPUT) real(stnd) On exit, TAU contains the value tau of the Householder reflector.

Further Details

On output, H is the identity matrix if TAU = 0.

This subroutine is adapted from:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Hanson, R.J., and Hopkins, T., 2017:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24 (December 2017), 23 pages.

6.7.12 `subroutine h2 ( beta, u, up, tau, piv, vec )`

Purpose

H2 generates a real elementary reflector H of order n, such that

H * X = D , with H' * H = I , X' = (alpha x) and D' = (beta 0)

where alpha and beta are scalars, and x is an (n-1)-element real vector.

H is represented in the form

H = I + tau * (v * v') ,

where tau is a real scalar and v is a real n-element vector.

The real elementary reflector H is then applied to a real vector C .

Arguments

BETA (INPUT/OUTPUT) real(stnd) On entry, the value alpha.

On exit, it is overwritten with the value beta.

U (INPUT/OUTPUT) real(stnd), dimension(:) On entry, U contains the pivot (n-1)-element vector x. On exit, U contains the vector v(2:) of the Householder reflector.

UP (OUTPUT) real(stnd) On exit, UP contains the value v(1) of the Householder reflector.

TAU (OUTPUT) real(stnd) On exit, TAU contains the value tau of the Householder reflector.

PIV (INPUT/OUTPUT) real(stnd) On entry, the scalar C[1].

On exit, PIV is overwritten by the scalar (H * C)[1].

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the real vector C[2:].

On exit, C is overwritten by the vector (H * C)[2:].

Further Details

On output, H is the identity matrix if TAU = 0.

It is assumed that size(VEC) >= size(U) .

This subroutine is adapted from:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Hanson, R.J., and Hopkins, T., 2017:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24 (December 2017), 23 pages.

6.7.13 `subroutine h2 ( beta, u, up, tau, vec_piv, mat, left )`

Purpose

H2 generates a real elementary reflector H of order n, such that

H * X = D , with H' * H = I , X' = (alpha x) and D' = (beta 0)

where alpha and beta are scalars, and x is an (n-1)-element real vector.

H is represented in the form

H = I + tau * (v * v') ,

where tau is a real scalar and v is a real n-element vector.

The real elementary reflector H is then applied to a real matrix C from the left or the right.

Arguments

BETA (INPUT/OUTPUT) real(stnd) On entry, the value alpha.

On exit, it is overwritten with the value beta.

U (INPUT/OUTPUT) real(stnd), dimension(:) On entry, U contains the pivot (n-1)-element vector x. On exit, U contains the vector v(2:) of the Householder reflector.

UP (OUTPUT) real(stnd) On exit, UP contains the value v(1) of the Householder reflector.

TAU (OUTPUT) real(stnd) On exit, TAU contains the value tau of the Householder reflector.

VEC_PIV (INPUT/OUTPUT) real(stnd), dimension(:) If LEFT=true:

- On entry, the row_vector C[1,:];
- On exit, VEC_PIV is overwritten by the vector (H * C)[1,:].

If LEFT=false:

- On entry, the column_vector C[:,1];
- On exit, VEC_PIV is overwritten by the vector (C * H)[:,1].

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) If LEFT=true:

- On entry, the (n-1)-by-m matrix C[2:,:];
- On exit, MAT is overwritten by the matrix (H * C)[2:,:] .

If LEFT=false:

- On entry, the m-by-(n-1) matrix C[:,2:];
- On exit, MAT is overwritten by the matrix (C * H)[:,2:].

LEFT (INPUT) logical(lgl) If:

- LEFT=true, H is applied to the real matrix C from the left;
- LEFT=false, H is applied to the real matrix C from the right.

Further Details

On output, H is the identity matrix if TAU = 0.

It is assumed that:

- size(MAT, 1) >= size(U) and size(MAT, 2) >= size(VEC_PIV) if LEFT=true;
- size(MAT, 2) >= size(U) and size(MAT, 1) >= size(VEC_PIV) if LEFT=false.

This subroutine is adapted from:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Hanson, R.J., and Hopkins, T., 2017:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24 (December 2017), 23 pages.

6.7.14 `subroutine apply_h1 ( u, tau, vec )`

Purpose

APPLY_H1 applies a real elementary reflector H generated by H1 to a real vector C . H is represented in the form

H = I + tau * (v * v') ,

where tau is a real scalar and v is a real n-element vector.

Arguments

U (INPUT) real(stnd), dimension(:) On entry, U contains the vector v of the Householder reflector, as generated by H1.

TAU (INPUT) real(stnd) On entry, the scalar tau of the Householder reflector, as generated by H1.

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the real vector C .

On exit, C is overwritten by the vector H * C .

Further Details

It is assumed that size(VEC) >= size(U) .

This subroutine is adapted from

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

6.7.15 subroutine apply_h1 (u, tau, mat, left)

Purpose

APPLY_H1 applies a real elementary reflector H generated by H1 to a real n-by-m or m-by-n matrix, C, from the left or the right. H is represented in the form

H = I + tau * (v * v') ,

where tau is a real scalar and v is a real n-element vector.

Arguments

U (INPUT) real(stnd), dimension(:) On entry, U contains the vector v of the Householder reflector, as generated by H1.

TAU (INPUT) real(stnd) On entry, the scalar tau of the Householder reflector, as generated by H1.

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the n-by-m or m-by-n matrix C.

On exit, C is overwritten by the matrix H * C (LEFT=true) or C * H (LEFT=false).

LEFT (INPUT) logical(lgl) If:

- LEFT=true, H is applied to the real matrix C from the left;
- LEFT=false, H is applied to the real matrix C from the right.

Further Details

It is assumed that:

- size(MAT, 1) >= size(U) if LEFT=true;
- size(MAT, 2) >= size(U) if LEFT=false.

This subroutine is adapted from:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

6.7.16 subroutine apply_h2 (u, up, tau, piv, vec)

Purpose

APPLY_H2 applies a real elementary reflector H generated by H2 to a real vector C . H is represented in the form

H = I + tau * (v * v') ,

where tau is a real scalar and v is a real n-element vector.

Arguments

U (INPUT) real(stnd), dimension(:) On entry, U contains the vector v(2:) of the Householder reflector, as generated by H2.

UP (INPUT) real(stnd) On entry, the value v(1) of the Householder reflector, as generated by H2.

TAU (INPUT) real(stnd) On entry, the scalar tau of the Householder reflector, as generated by H2.

PIV (INPUT/OUTPUT) real(stnd) On entry, the scalar C[1].

On exit, PIV is overwritten by the scalar (H * C)[1].

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the real vector C[2:].

On exit, C is overwritten by the vector (H * C)[2:].

Further Details

It is assumed that size(VEC) >= size(U) .

This subroutine is adapted from:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

6.7.17 subroutine apply_h2 (u, up, tau, vec_piv, mat, left)

Purpose

APPLY_H2 applies a real elementary reflector H generated by H2 to a real n-by-m or m-by-n matrix, C, from the left or the right. H is represented in the form

H = I + tau * (v * v') ,

where tau is a real scalar and v is a real n-element vector.

Arguments

U (INPUT) real(stnd), dimension(:) On entry, U contains the vector v(2:) of the Householder reflector, as generated by H2.

UP (INPUT) real(stnd) On entry, the value v(1) of the Householder reflector, as generated by H2.

TAU (INPUT) real(stnd) On entry, the scalar tau of the Householder reflector, as generated by H2.

VEC_PIV (INPUT/OUTPUT) real(stnd), dimension(:) If LEFT=true:

- On entry, the row_vector C[1,:];
- On exit, VEC_PIV is overwritten by the vector (H * C)[1,:].

If LEFT=false:

- On entry, the column_vector C[:,1];
- On exit, VEC_PIV is overwritten by the vector (C * H)[:,1].

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) If LEFT=true:

- On entry, the (n-1)-by-m matrix C[2:,:];
- On exit, MAT is overwritten by the matrix (H * C)[2:,:] .

If LEFT=false:

- On entry, the m-by-(n-1) matrix C[:,2:];
- On exit, MAT is overwritten by the matrix (C * H)[:,2:].

LEFT (INPUT) logical(lgl) If:

- LEFT=true, H is applied to the real matrix C from the left;
- LEFT=false, H is applied to the real matrix C from the right.

Further Details

It is assumed that:

- size(MAT, 1) >= size(U) and size(MAT, 2) >= size(VEC_PIV) if LEFT=true;
- size(MAT, 2) >= size(U) and size(MAT, 1) >= size(VEC_PIV) if LEFT=false.

This subroutine is adapted from:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

6.8 Module_LLSQ_Procedures

Copyright 2025 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING SUBROUTINES AND FUNCTIONS FOR SOLVING LINEAR LEAST SQUARES PROBLEMS.

LATEST REVISION : 08/09/2025

6.8.1 `function solve_llsq ( a, b, krank, tol, min_norm )`

Purpose

SOLVE_LLSQ computes a solution to a real linear least squares problem:

Minimize 2-norm || B - A * X ||

using an orthogonal factorization with columns pivoting of A. A is a m-by-n matrix which may be rank-deficient. m>=n or n>m is permitted. Here, B is a m-element right hand side vector and X is a n-element solution vector.

The function returns the n-element solution vector X.

A and B are not overwritten by SOLVE_LLSQ.

Arguments

A (INPUT) real(stnd), dimension(:,:) On entry, the m-by-n coefficient matrix A.

B (INPUT) real(stnd), dimension(:) On entry, the m-element right hand side vector B.

The shape of B must verify:

- size(B) = size(A, 1) = m .

KRANK (INPUT, OPTIONAL) integer(i4b) On entry, KRANK=k implies that the first k columns of A are to be forced into the basis, pivoting is performed on the last n-k columns of A.

When KRANK >=min(m,n) is used, pivoting is not performed. This is appropriate when A is known to be full rank.

If KRANK is absent or is <=0, pivoting is done on all columns of A. This is appropriate if A may not be of full rank (i.e. certain columns of A are linear combinations of other columns).

TOL (INPUT, OPTIONAL) real(stnd) On entry, TOL is used to determine the effective rank of A, which is then defined as the order of the largest leading triangular submatrix R11 in the QR factorization (with pivoting) of A whose estimated condition number, in the 1-norm, is less than 1/TOL. TOl must be set to the relative precision of the elements in A and B. If each element is correct to, say, 5 digits then TOL=0.00001 should be used.

TOL must not be greater or equal to 1 or less or equal than 0, otherwise the numerical rank of A is determined and the calculations to determine the condition number are not performed. If TOL is absent, the numerical rank of A is determined.

MIN_NORM (INPUT, OPTIONAL) logical(lgl) On entry, if:

- MIN_NORM=true, the minimun 2-norm solution is computed.
- MIN_NORM=false or if MIN_NORM is absent, a solution is computed such that if the j-th column of A is omitted from the basis, X[j] is set to zero.

Further Details

1) The routine first computes a QR factorization with (partial) column pivoting on option (see below):

   A * P = Q * R

, here P is n-by-n permutation matrix, R is an upper triangular or trapezoidal (if n>m) matrix and Q is a m-by-m orthogonal matrix.

R can then be partioned by defining R11 as the largest leading submatrix of R whose estimated condition number, in the 1-norm, is less than 1/TOL (or such that abs(R[j,j])>0 if TOL is absent). The order of R11, arank, is the effective rank of A.

This leads to the following partition of R:

[R11 R12]

[R21 R22]

where R21 is zero by construction (since R is an upper triangular or trapezoidal) and R22 is considered to be negligible.

2) If MIN_NORM is present and has the value true, R12 is annihilated by orthogonal transformations from the right, arriving at the complete orthogonal factorization:

   A * P = Q * T * Z

   , where Z is a n-by-n orthogonal matrix and T has the form:

   [T11 T12]

   [T21 T22]

   , here T21 (=R21), T12 and T22 (=R22) are zero and T11 is a arank-by-arank upper triangular matrix.

   The minimum 2-norm solution is then

   X = [P * Z'](:,:arank) * [inv(T11) * Q1' * B]

   where inv(T11) is the inverse of T11, Z' is the transpose of Z and Q1 consists of the first arank columns of Q.

3) If MIN_NORM is absent or has the value false, a solution is computed as

   X = P(:,:arank) * [inv(R11) * Q1' * B]

   where inv(R11) is the inverse of R11 and Q1 consists of the first arank columns of Q. In this case, if the j-th column of A is omitted from the basis, X[j] is set to zero.

4) On input, if KRANK is present and KRANK=k, the first k columns of A are to be forced into the basis. Pivoting is performed on the last n-k columns of A.

   When KRANK is present and KRANK>=min(m,n) is used, pivoting is not performed. This is appropriate when A is known to be of full rank.

   If KRANK is absent or is present with KRANK<=0, pivoting is done on all columns of A.

5) TOL is an optional argument such that 0<TOL<1. If TOL is not specified, or is outside]0,1[, the calculations to determine the condition number of A are not performed and crude tests on R(j,j) are performed to determine the numerical rank of A. If TOL is present and is in]0,1[, the calculations to determine the condition number are performed.

6) If it is possible that A may not be full rank (i.e., certain columns of A are linear combinations of other columns), then the linearly dependent columns can usually be eliminated by using KRANK=0 and TOL=relative precision of the elements in A and B. If each element is correct to, say, 5 digits then TOL=0.00001 should be used. Also, it may be helpful to scale the columns of A so that all elements are about the same order of magnitude.

7) On exit, if A or B are empty, the function returns a n-element vector filled with nan() value.

For further details on linear least square problems and algorithms to solve them, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Hansen, P.C., Pereyra, V., and Scherer, G., 2012:** Least Squares Data Fitting with Applications. Johns Hopkins University Press, 328 pp.

6.8.2 function solve_llsq (a, b, krank, tol, min_norm)

Purpose

SOLVE_LLSQ computes solutions to real linear least squares problems of the form:

Minimize 2-norm|| B - A * X ||

using an orthogonal factorization with columns pivoting of A. A is a m-by-n matrix which may be rank-deficient. m>=n or n>m is permitted.

Several right hand side vectors b can be handled in a single call; they are stored as the columns of the m-by-nb right hand side matrix B.

The function returns the n-by-nb solution matrix X.

A and B are not overwritten by SOLVE_LLSQ.

Arguments

A (INPUT) real(stnd), dimension(:,:) On entry, the m-by-n coefficient matrix A.

B (INPUT) real(stnd), dimension(:,:) On entry, the m-by-nb right hand side matrix B.

The shape of B must verify:

- size(B, 1) = size(A, 1) = m .

KRANK (INPUT, OPTIONAL) integer(i4b) On entry, KRANK=k implies that the first k columns of A are to be forced into the basis, pivoting is performed on the last n-k columns of A.

When KRANK >=min(m,n) is used, pivoting is not performed. This is appropriate when A is known to be full rank.

If KRANK is absent or is <=0, pivoting is done on all columns of A. This is appropriate if A may not be of full rank (i.e. certain columns of A are linear combinations of other columns).

TOL (INPUT, OPTIONAL) real(stnd) On entry, TOL is used to determine the effective rank of A, which is then defined as the order of the largest leading triangular submatrix R11 in the QR factorization (with pivoting) of A whose estimated condition number, in the 1-norm, is less than 1/TOL. TOl must be set to the relative precision of the elements in A and B. If each element is correct to, say, 5 digits then TOL=0.00001 should be used.

TOL must not be greater or equal to 1 or less or equal than 0, otherwise the numerical rank of A is determined and the calculations to determine the condition number are not performed. If TOL is absent, the numerical rank of A is determined.

MIN_NORM (INPUT, OPTIONAL) logical(lgl) On entry, if:

- MIN_NORM=true, the minimun 2-norm solutions are computed.
- MIN_NORM=false or if MIN_NORM is absent, solutions are computed such that if the j-th column of A is omitted from the basis, X[j,:] is set to zero.

Further Details

1) The routine first computes a QR factorization with (partial) column pivoting on option (see below):

A * P = Q * R

, here P is n-by-n permutation matrix, R is an upper triangular or trapezoidal (if n>m) matrix and Q is a m-by-m orthogonal matrix.

R can then be partioned by defining R11 as the largest leading submatrix of R whose estimated condition number, in the 1-norm, is less than 1/TOL (or such that abs(R[j,j])>0 if TOL is absent). The order of R11, arank, is the effective rank of A.

This leads to the following partition of R:

[R11 R12]

[R21 R22]

where R21 is zero by construction (since R is an upper triangular or trapezoidal) and R22 is considered to be negligible.

2) If MIN_NORM is present and has the value true, R12 is annihilated by orthogonal transformations from the right, arriving at the complete orthogonal factorization:

    A * P = Q * T * Z

, where Z is a n-by-n orthogonal matrix and T has the form:

[T11 T12]

[T21 T22]

, here T21 (=R21), T12 and T22 (=R22) are zero and T11 is a arank-by-arank upper triangular matrix.

The minimum 2-norm solution is then

    X = [P * Z'](:,:arank) * [inv(T11) * Q1' * B]

where inv(T11) is the inverse of T11, Z' is the transpose of Z and Q1 consists of the first arank columns of Q.

3) If MIN_NORM is absent or has the value false, a solution is computed as

    X = P(:,:arank) * [inv(R11) * Q1' * B]

where inv(R11) is the inverse of R11 and Q1 consists of the first arank columns of Q. In this case, if the j-th column of A is omitted from the basis, X[j] is set to zero.

4) On input, if KRANK is present and KRANK=k, the first k columns of A are to be forced into the basis. Pivoting is performed on the last n-k columns of A.

   When KRANK is present and KRANK>=min(m,n) is used, pivoting is not performed. This is appropriate when A is known to be full rank.

   If KRANK is absent or is present with KRANK<=0, pivoting is done on all columns of A.

5) TOL is an optional argument such that 0<TOL<1. If TOL is not specified, or is outside]0,1[, the calculations to determine the condition number of A are not performed and crude tests on R(j,j) are performed to determine the numerical rank of A. If TOL is present and is in]0,1[, the calculations to determine the condition number are performed.

6) If it is possible that A may not be full rank (i.e., certain columns of A are linear combinations of other columns), then the linearly dependent columns can usually be eliminated by using KRANK=0 and TOL=relative precision of the elements in A and B. If each element is correct to, say, 5 digits then TOL=0.00001 should be used. Also, it may be helpful to scale the columns of A so that all elements are about the same order of magnitude.

7) On exit, if A or B are empty, the function returns a n-by-nb matrix filled with nan() value.

For further details on linear least square problems and algorithms to solve them, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Hansen, P.C., Pereyra, V., and Scherer, G., 2012:** Least Squares Data Fitting with Applications. Johns Hopkins University Press, 328 pp.

6.8.3 `function solve_llsq ( a, b )`

Purpose

SOLVE_LLSQ computes a solution to a real linear least squares problem:

Minimize 2-norm|| B - A * X ||

A is a m-element vector, B is a m-element right hand side vector and X is a real scalar.

The function returns the solution scalar X.

A and B are not overwritten by SOLVE_LLSQ.

Arguments

A (INPUT) real(stnd), dimension(:) On entry, the m-element coefficient vector A.

B (INPUT) real(stnd), dimension(:) On entry, the m-element right hand side vector B.

The shape of B must verify:

- size(B) = size(A) = m .

Further Details

1) The routine first generates a real elementary reflector H of order m, such that

   H * A = D , with H' * H = I and D' = (d 0)

   where d is a scalar. H is represented in the form

   H = I + beta * (v * v') ,

   where beta is a real scalar and v is a real m-element vector.

2) The solution X is then computed as

   X = [H * B](1) / d

3) On exit, if A or B are empty, the function returns a nan() value.

6.8.4 `function solve_llsq ( a, b )`

Purpose

SOLVE_LLSQ computes solutions to real linear least squares problems of the form:

Minimize 2-norm|| B - A * X ||

A is a m-element vector and several right hand side vectors b can be handled in a single call; they are stored as the columns of the m-by-nb right hand side matrix B.

The function returns the nb-element solution vector X.

A and B are not overwritten by SOLVE_LLSQ.

Arguments

A (INPUT) real(stnd), dimension(:) On entry, the m-element coefficient vector A.

B (INPUT) real(stnd), dimension(:,:) On entry, the m-by-nb right hand side matrix B.

The shape of B must verify:

- size(B, 1) = size(A) = m .

Further Details

1) The routine first generates a real elementary reflector H of order m, such that

   H * A = D , with H' * H = I and D' = (d 0)

   where d is a scalar. H is represented in the form

   H = I + beta * (v * v') ,

   where beta is a real scalar and v is a real m-element vector.

2) The solution vector X is then computed as

   X(:) = [H * B](1,:) / d

3) On exit, if A or B are empty, the function returns a nb-vector filled with nan() value.

6.8.5 subroutine llsq_qr_solve (mat, b, x, rnorm, resid, krank, tol, min_norm)

Purpose

LLSQ_QR_SOLVE computes a solution to a real linear least squares problem:

Minimize 2-norm|| B - MAT * X ||

using an orthogonal factorization with columns pivoting of MAT. MAT is a m-by-n matrix which may be rank-deficient. m>=n or n>m is permitted. Here, B is a m-element right hand side vector and X is a n-element solution vector.

MAT and B are not overwritten by LLSQ_QR_SOLVE.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the m-by-n coefficient matrix MAT.

B (INPUT) real(stnd), dimension(:) On entry, the m-element right hand side vector B.

The shape of B must verify:

- size(B) = size(MAT, 1) = m .

X (OUTPUT) real(stnd), dimension(:) On exit, the n-element solution vector X.

The shape of X must verify:

- size(X) = size(MAT, 2) = n .

RNORM (OUTPUT, OPTIONAL) real(stnd) On exit, the 2-norm of the residual vector for the solution vector X.

RESID (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the m-element residual vector for the solution vector X, RESID = B - MAT * X.

The shape of RESID must verify:

- size(RESID) = size(B) = size(MAT, 1) = m .

KRANK (INPUT/OUTPUT, OPTIONAL) integer(i4b) On entry, KRANK=k implies that the first k columns of MAT are to be forced into the basis, pivoting is performed on the last n-k columns of MAT.

When KRANK >=min(m,n) is used, pivoting is not performed. This is appropriate when MAT is known to be full rank.

If KRANK is absent or is <=0, pivoting is done on all columns of MAT. This is appropriate if MAT may not be of full rank (i.e. certain columns of MAT are linear combinations of other columns).

On exit, KRANK contains the effective rank of MAT, i.e., the number of independent columns in matrix MAT.

TOL (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, TOL is used to determine the effective rank of MAT, which is then defined as the order of the largest leading triangular submatrix R11 in the QR factorization (with pivoting) of MAT whose estimated condition number, in the 1-norm, is less than 1/TOL. TOl must be set to the relative precision of the elements in MAT and B. If each element is correct to, say, 5 digits then TOL=0.00001 should be used.

TOL must not be greater or equal to 1 or less than 0, otherwise the numerical rank of MAT is determined and the calculations to determine the condition number are not performed. If TOL=0, the numerical rank of MAT is determined, but the condition number is calculated.

On exit, if a condition number is calculated, its reciprocal is returned in TOL. Otherwise, TOL is not changed.

If TOL is absent, the numerical rank of MAT is used and is returned in the optional argument KRANK.

MIN_NORM (INPUT, OPTIONAL) logical(lgl) On entry, if:

- MIN_NORM=true, the minimun 2-norm solution is computed.
- MIN_NORM=false or if MIN_NORM is absent, a solution is computed such that if the j-th column of A is omitted from the basis, X[j] is set to zero.

Further Details

1) The routine first computes a QR factorization with (partial) column pivoting on option (see below):

   MAT * P = Q * R

, here P is n-by-n permutation matrix, R is an upper triangular or trapezoidal (if n>m) matrix and Q is a m-by-m orthogonal matrix.

R can then be partioned by defining R11 as the largest leading submatrix of R whose estimated condition number, in the 1-norm, is less than 1/TOL (or such that abs(R[j,j])>0 if TOL is absent). The order of R11, KRANK, is the effective rank of MAT.

This leads to the following partition of R:

[R11 R12]

[R21 R22]

where R21 is zero by construction (since R is an upper triangular or trapezoidal) and R22 is considered to be negligible.

2) If MIN_NORM is present and has the value true, R12 is annihilated by orthogonal transformations from the right, arriving at the complete orthogonal factorization:

   MAT * P = Q * T * Z

, where Z is a n-by-n orthogonal matrix and T has the form:

[T11 T12]

[T21 T22]

, here T21 (=R21), T12 and T22 (=R22) are zero and T11 is a KRANK-by-KRANK upper triangular matrix.

The minimum 2-norm solution is then

   X = [P * Z'](:,:KRANK) * [inv(T11) * Q1' * B]

where inv(T11) is the inverse of T11, Z' is the transpose of Z and Q1 consists of the first KRANK columns of Q.

3) If MIN_NORM is absent or has the value false, a solution is computed as

   X = P(:,:KRANK) * [inv(R11) * Q1' * B]

where inv(R11) is the inverse of R11 and Q1 consists of the first KRANK columns of Q. In this case, if the j-th column of MAT is omitted from the basis, X[j] is set to zero.

4) In both cases:

   - The 2-norm of the residual vector for the solution X can be obtained through the optional argument RNORM .
   - The m-element residual vector, B - MAT * X, can be obtained through the optional argument RESID.

5) On input, if KRANK is present and KRANK=k, the first k columns of MAT are to be forced into the basis. Pivoting is performed on the last n-k columns of MAT.

   When KRANK is present and KRANK>=min(m,n) is used, pivoting is not performed. This is appropriate when MAT is known to be full rank.

   If KRANK is absent or is present with KRANK<=0, pivoting is done on all columns of MAT.

   On output, if KRANK is present, it contains the effective rank of MAT, i.e., the order of the submatrix R11.

6) TOL is an optional argument such that 0<=TOL<1. If TOL is not specified, or is outside [0,1[, the calculations to determine the condition number of MAT are not performed and crude tests on R(j,j) are performed to determine the numerical rank of MAT. If TOL is present and is in [0,1[, the calculations to determine the condition number are performed and its reciprocal is return in TOL.

7) If it is possible that MAT may not be full rank (i.e., certain columns of MAT are linear combinations of other columns), then the linearly dependent columns can usually be eliminated by using KRANK=0 and TOL=relative precision of the elements in MAT and B. If each element is correct to, say, 5 digits then TOL=0.00001 should be used. Also, it may be helpful to scale the columns of MAT so that all elements are about the same order of magnitude.

For further details on linear least square problems and algorithms to solve them, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Hansen, P.C., Pereyra, V., and Scherer, G., 2012:** Least Squares Data Fitting with Applications. Johns Hopkins University Press, 328 pp.

6.8.6 subroutine llsq_qr_solve (mat, b, x, rnorm, resid, krank, tol, min_norm)

Purpose

LLSQ_QR_SOLVE computes solutions to real linear least squares problems of the form:

Minimize 2-norm|| B - MAT * X ||

using an orthogonal factorization with columns pivoting of MAT. MAT is a m-by-n matrix which may be rank-deficient. m>=n or n>m is permitted.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m-by-nb right hand side matrix B and the n-by-nb solution matrix X, respectively.

MAT and B are not overwritten by LLSQ_QR_SOLVE.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the m-by-n coefficient matrix MAT.

B (INPUT) real(stnd), dimension(:,:) On entry, the m-by-nb right hand side matrix B.

The shape of B must verify:

- size(B, 1) = size(MAT, 1) = m
- size(B, 2) = size(X, 2) = nb .

X (OUTPUT) real(stnd), dimension(:,:) On exit, the n-by-nb solution matrix X.

The shape of X must verify:

- size(X, 1) = size(MAT, 2) = n
- size(X, 2) = size(B, 2) = nb .

RNORM (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the 2-norm of the residual vectors for the solutions stored columnwise in the matrix X.

The size of RNORM must verify:

- size(RNORM) = size(X, 2) = size(B, 2) = nb .

RESID (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, the residual vectors for the solutions stored columnwise in the matrix X, RESID = B - MAT * X.

The shape of RESID must verify:

- size(RESID, 1) = size(B, 1) = size(MAT, 1) = m
- size(RESID, 2) = size(B, 2) = size(X, 2) = nb .

KRANK (INPUT/OUTPUT, OPTIONAL) integer(i4b) On entry, KRANK=k implies that the first k columns of MAT are to be forced into the basis, pivoting is performed on the last n-k columns of MAT.

When KRANK >=min(m,n) is used, pivoting is not performed. This is appropriate when MAT is known to be full rank.

If KRANK is absent or is <=0, pivoting is done on all columns of MAT. This is appropriate if MAT may not be of full rank (i.e. certain columns of MAT are linear combinations of other columns).

On exit, KRANK contains the effective rank of MAT, i.e., the number of independent columns in matrix MAT.

TOL (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, TOL is used to determine the effective rank of MAT, which is then defined as the order of the largest leading triangular submatrix R11 in the QR factorization (with pivoting) of MAT whose estimated condition number, in the 1-norm, is less than 1/TOL. TOl must be set to the relative precision of the elements in MAT and B. If each element is correct to, say, 5 digits then TOL=0.00001 should be used.

TOL must not be greater or equal to 1 or less than 0, otherwise the numerical rank of MAT is determined and the calculations to determine the condition number are not performed. If TOL=0, the numerical rank of MAT is determined, but the condition number is calculated.

On exit, if a condition number is calculated, its reciprocal is returned in TOL. Otherwise, TOL is not changed.

If TOL is absent, the numerical rank of MAT is used and is returned in the optional argument KRANK.

MIN_NORM (INPUT, OPTIONAL) logical(lgl) On entry, if:

- MIN_NORM=true, the minimun 2-norm solutions are computed.
- MIN_NORM=false or if MIN_NORM is absent, solutions are computed such that if the j-th column of A is omitted from the basis, X[j,:] is set to zero.

Further Details

1) The routine first computes a QR factorization with (partial) column pivoting on option (see below):

   MAT * P = Q * R

, here P is n-by-n permutation matrix, R is an upper triangular or trapezoidal (if n>m) matrix and Q is a m-by-m orthogonal matrix.

R can then be partioned by defining R11 as the largest leading submatrix of R whose estimated condition number, in the 1-norm, is less than 1/TOL (or such that abs(R[j,j])>0 if TOL is absent). The order of R11, KRANK, is the effective rank of MAT.

This leads to the following partition of R:

[R11 R12]

[R21 R22]

where R21 is zero by construction (since R is an upper triangular or trapezoidal) and R22 is considered to be negligible.

2) If MIN_NORM is present and has the value true, R12 is annihilated by orthogonal transformations from the right, arriving at the complete orthogonal factorization:

   MAT * P = Q * T * Z

   , where Z is a n-by-n orthogonal matrix and T has the form:

   [T11 T12]

   [T21 T22]

   , here T21 (=R21), T12 and T22 (=R22) are zero and T11 is a KRANK-by-KRANK upper triangular matrix.

   The minimum 2-norm solution is then

   X = [P * Z'](:,:KRANK) * [inv(T11) * Q1' * B]

   where inv(T11) is the inverse of T11, Z' is the transpose of Z and Q1 consists of the first KRANK columns of Q.

3) If MIN_NORM is absent or has the value false, a solution is computed as

   X = P(:,:KRANK) * [inv(R11) * Q1' * B]

   where inv(R11) is the inverse of R11 and Q1 consists of the first KRANK columns of Q. In this case, if the j-th column of MAT is omitted from the basis, X[j] is set to zero.

4) In both cases:

   - The 2-norm of the residual vector for the solution in the j-th column of X is given in RNORM[j] if argument RNORM is present.
   - The residual matrix, B - MAT * X, can be obtained through the optional argument RESID.

5) On input, if KRANK is present and KRANK=k, the first k columns of MAT are to be forced into the basis. Pivoting is performed on the last n-k columns of MAT.

   When KRANK is present and KRANK>=min(m,n) is used, pivoting is not performed. This is appropriate when MAT is known to be full rank.

   If KRANK is absent or is present with KRANK<=0, pivoting is done on all columns of MAT.

   On output, if KRANK is present, it contains the effective rank of MAT, i.e., the order of the submatrix R11.

6) TOL is an optional argument such that 0<=TOL<1. If TOL is not specified, or is outside [0,1[, the calculations to determine the condition number of MAT are not performed and crude tests on R(j,j) are performed to determine the numerical rank of MAT. If TOL is present and is in [0,1[, the calculations to determine the condition number are performed and its reciprocal is return in TOL.

7) If it is possible that MAT may not be full rank (i.e., certain columns of MAT are linear combinations of other columns), then the linearly dependent columns can usually be eliminated by using KRANK=0 and TOL=relative precision of the elements in MAT and B. If each element is correct to, say, 5 digits then TOL=0.00001 should be used. Also, it may be helpful to scale the columns of MAT so that all elements are about the same order of magnitude.

For further details on linear least square problems and algorithms to solve them, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Hansen, P.C., Pereyra, V., and Scherer, G., 2012:** Least Squares Data Fitting with Applications. Johns Hopkins University Press, 328 pp.

6.8.7 `subroutine llsq_qr_solve ( vec, b, x, rnorm, resid )`

Purpose

LLSQ_QR_SOLVE computes a solution to a real linear least squares problem:

Minimize 2-norm|| B - VEC * X ||

VEC is a m-element vector, B is a m-element right hand side vector and X is a real scalar.

VEC and B are not overwritten by LLSQ_QR_SOLVE.

Arguments

VEC (INPUT) real(stnd), dimension(:) On entry, the m-element coefficient vector VEC.

B (INPUT) real(stnd), dimension(:) On entry, the m-element right hand side vector B.

The shape of B must verify:

- size(B) = size(VEC) = m .

X (OUTPUT) real(stnd) On exit, the real solution X.

RNORM (OUTPUT, OPTIONAL) real(stnd) On exit, the 2-norm of the residual vector for the solution scalar X.

RESID (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the m-element residual vector for the solution X, RESID = B - MAT * X.

The shape of RESID must verify:

- size(RESID) = size(B) = size(VEC) = m .

Further Details

1) The routine first generates a real elementary reflector H of order m, such that

   H * VEC = D , with H' * H = I and D' = (d 0)

   where d is a scalar. H is represented in the form

   H = I + beta * (v * v') ,

   where beta is a real scalar and v is a real m-element vector.

2) The solution X is then computed as

   X = [H * B](1) / d

3) The 2-norm of the residual vector for the solution X can be obtained through the optional argument RNORM as

   2-norm|| [H * B](2:) ||

4) The residual vector, B - VEC * X, can be obtained through the optional argument RESID.

6.8.8 subroutine llsq_qr_solve (vec, b, x, rnorm, resid)

Purpose

LLSQ_QR_SOLVE computes solutions to real linear least squares problems of the form:

Minimize 2-norm|| B - VEC * X ||

VEC is a m-element vector and several right hand side vectors b and solution scalars x can be handled in a single call; they are stored as the columns of the m-by-nb right hand side matrix B and the nb-element solution vector X, respectively.

VEC and B are not overwritten by LLSQ_QR_SOLVE.

Arguments

VEC (INPUT) real(stnd), dimension(:) On entry, the m-element coefficient vector VEC.

B (INPUT) real(stnd), dimension(:,:) On entry, the m-by-nb right hand side matrix B.

The shape of B must verify:

- size(B, 1) = size(VEC) = m
- size(B, 2) = size(X) = nb .

X (OUTPUT) real(stnd), dimension(:) On exit, the nb-element solution vector X.

The shape of X must verify: size(X) = size(B, 2) = nb .

RNORM (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the 2-norms of the residual vectors for the solutions stored in the vector X.

The size of RNORM must verify:

- size(RNORM) = size(X) = size(B, 2) = nb .

RESID (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, the residual vectors for the solutions stored in the vector X, RESID = B - VEC * X.

The shape of RESID must verify:

- size(RESID, 1) = size(B, 1) = size(VEC) = m
- size(RESID, 2) = size(B, 2) = size(X) = nb .

Further Details

1) The routine first generates a real elementary reflector H of order m, such that

   H * A = D , with H' * H = I and D' = (d 0)

   where d is a scalar. H is represented in the form

   H = I + beta * (v * v') ,

   where beta is a real scalar and v is a real m-element vector.

2) The solution vector X is then computed as

   X(:) = [H * B](1,:) / d

3) The 2-norm of the residual vector for the solution X[j] is given in RNORM[j] if argument RNORM is present.

4) The residual matrix, B - VEC * X, can be obtained through the optional argument RESID.

6.8.9 subroutine llsq_qr_solve2 (mat, b, x, rnorm, comp_resid, krank, tol, min_norm, diagr, beta, ip, tau)

Purpose

LLSQ_QR_SOLVE2 computes a solution to a real linear least squares problem:

Minimize 2-norm|| B - MAT * X ||

using a (complete) orthogonal factorization of MAT. MAT is a m-by-n matrix which may be rank-deficient. m>=n or n>m is permitted. Here, B is a m-element right hand side vector and X is a n-element solution vector.

MAT and B are overwritten with information generated by LLSQ_QR_SOLVE2. The (complete) orthogonal factorization of MAT is saved in arguments MAT, DIAGR, BETA, IP and TAU on output.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n coefficient matrix MAT.

On exit, MAT has been overwritten by details of its (complete) orthogonal factorization. Other parts of the factorization can be obtained if the optional arguments DIAGR, BETA, IP and TAU are present.

See Further Details.

B (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the m-element right hand side vector B.

On exit, if COMP_RESID is present and is equal true, the residual vector B - MAT * X overwrites B on output.

The shape of B must verify:

- size(B) = size(MAT, 1) = m .

X (OUTPUT) real(stnd), dimension(:) On exit, the n-element solution vector X.

The shape of X must verify:

- size(X) = size(MAT, 2) = n .

RNORM (OUTPUT, OPTIONAL) real(stnd) On exit, the 2-norm of the residual vector for the solution vector X.

COMP_RESID (INPUT, OPTIONAL) logical(lgl) On entry, if COMP_RESID is present and is equal true, the residual vector B - MAT * X overwrites B on exit.

KRANK (INPUT/OUTPUT, OPTIONAL) integer(i4b) On entry, KRANK=k implies that the first k columns of MAT are to be forced into the basis, pivoting is performed on the last n-k columns of MAT.

When KRANK >=min(m,n) is used, pivoting is not performed. This is appropriate when MAT is known to be full rank.

If KRANK is absent or is <=0, pivoting is done on all columns of MAT. This is appropriate if MAT may not be of full rank (i.e. certain columns of MAT are linear combinations of other columns).

On exit, KRANK contains the effective rank of MAT, i.e., the number of independent columns in matrix MAT.

TOL (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, TOL is used to determine the effective rank of MAT, which is then defined as the order of the largest leading triangular submatrix R11 in the QR factorization (with pivoting) of MAT whose estimated condition number, in the 1-norm, is less than 1/TOL. TOL must be set to the relative precision of the elements in MAT and B. If each element is correct to, say, 5 digits then TOL=0.00001 should be used.

TOL must not be greater or equal to 1 or less than 0, otherwise the numerical rank of MAT is determined and the calculations to determine the condition number are not performed. If TOL=0, the numerical rank of MAT is determined, but the condition number is calculated.

On exit, if a condition number is calculated, its reciprocal is returned in TOL. Otherwise, TOL is not changed.

If TOL is absent, the numerical rank of MAT is used and is returned in the optional argument KRANK.

MIN_NORM (INPUT, OPTIONAL) logical(lgl) On entry, if:

- MIN_NORM=true, a complete orthogonal factorization of MAT and the minimun 2-norm solution is computed.
- MIN_NORM=false or if MIN_NORM is absent, a QR factorization with column pivoting of MAT and a solution is computed such that if the j-th column of MAT is omitted from the basis, X[j] is set to zero.

DIAGR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On entry, if:

- MIN_NORM=false or is absent, the diagonal elements of the matrix R.
- MIN_NORM=true, the diagonal elements of the matrix T11. The diagonal elements of T11 are stored in DIAGR(1:KRANK).

See Further Details.

The size of DIAGR must be min(size(MAT,1) , size(MAT,2)).

BETA (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the scalar factors of the elementary reflectors defining Q.

See Further Details.

The size of BETA must be min(size(MAT,1) , size(MAT,2)).

IP (OUTPUT, OPTIONAL) integer(i4b), dimension(:) On exit, if IP(j)=k, then the j-th column of MAT*P was the k-th column of MAT.

See Further Details.

The size of IP must be size(MAT,2) = n.

TAU (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the scalars factors of the elementary reflectors defining Z in the complete orthogonal factorization of MAT if MIN_NORM is present and is equal to true. Otherwise, TAU is set to 0.

See Further Details.

The size of TAU must be min(size(MAT,1) , size(MAT,2)).

Further Details

1) The routine first computes a QR factorization with (partial) column pivoting on option (see below):

MAT * P = Q * R

, here P is a n-by-n permutation matrix, R is an upper triangular or trapezoidal (if n>m) matrix and Q is a m-by-m orthogonal matrix.

The matrix Q is represented as a product of elementary reflectors

Q = H(1) * H(2) * . . . * H(k), where k = min(size(MAT,1) , size(MAT,2))

Each H(i) has the form

H(i) = I + beta * (v * v') ,

where beta is a real scalar and v is a real m-element vector with v(1:i-1) = 0. v(i:m) is stored on exit in MAT(i:m,i) and beta in BETA(i).

The matrix P is represented in the array IP as follows: If IP(j) = i then the jth column of P is the ith canonical unit vector.

The elements above the diagonal of the array MAT contain the corresponding elements of the triangular matrix R. The elements on the diagonal of R are stored in the array DIAGR.

R can then be partioned by defining R11 as the largest leading submatrix of R whose estimated condition number, in the 1-norm, is less than 1/TOL (or such that abs(R[j,j])>0 if TOL is absent). The order of R11, KRANK, is the effective rank of MAT.

This leads to the following partition of R:

[R11 R12]

[R21 R22]

where R21 is zero by construction (since R is an upper triangular or trapezoidal) and R22 is considered to be negligible.

2) If MIN_NORM is present and has the value true, R12 is annihilated by orthogonal transformations from the right, arriving at the complete orthogonal factorization:

MAT * P = Q * T * Z

, where Z is a n-by-n orthogonal matrix and T is a m-by-n matrix has the form:

[T11 T12]

[T21 T22]

, here T21 (=R21), T12 and T22 (=R22) are zero and T11 is a KRANK-by-KRANK upper triangular matrix.

The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), which is used to introduce zeros into the kth row of R, is given in the form

[I 0]

[0 T(k)]

where

T(k) = I + tau * (u(k) * u(k)') and u(k)' = (1 0 z(k))

tau is a scalar, u(k) is a n-k+1 vector and z(k) is an (n-KRANK) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of R12.

The Z n-by-n orthogonal matrix is given by

Z = Z(1) * Z(2) * . . . * Z(KRANK)

On exit, the scalar tau is returned in the kth element of TAU and the vector u(k) in the kth row of MAT, such that the elements of z(k) are in MAT(k,KRANK+1:n).

On exit, the elements above the diagonal of the array section MAT(1:KRANK,1:KRANK) contain the corresponding elements of the triangular matrix T11. The elements of the diagonal of T11 are stored in the array section DIAGR(1:KRANK). The last part of DIAGR is set to zero. In other words, T11 overwrites R11 and Z overwrites R12 on exit.

The minimum 2-norm solution is then

X = [P * Z'](:,:KRANK) * [inv(T11) * Q1' * B]

where inv(T11) is the inverse of T11, Z' is the transpose of Z and Q1 consists of the first KRANK columns of Q.

3) If MIN_NORM is absent or has the value false, a solution is computed as

X = P(:,:KRANK) * [inv(R11) * Q1' * B]

where inv(R11) is the inverse of R11 and Q1 consists of the first KRANK columns of Q. In this case, if the j-th column of MAT is omitted from the basis, X[j] is set to zero and R is not destroyed in MAT.

4) In both cases:

- The 2-norm of the residual vector for the solution X can be obtained through the optional argument RNORM .
- If COMP_RESID=true, The m-element residual vector B - MAT * X overwrites B on exit.

5) On input, if KRANK is present and KRANK=k, the first k columns of MAT are to be forced into the basis. Pivoting is performed on the last n-k columns of MAT.

When KRANK is present and KRANK>=min(m,n) is used, pivoting is not performed. This is appropriate when MAT is known to be full rank.

If KRANK is absent or is present with KRANK<=0, pivoting is done on all columns of MAT.

On output, if KRANK is present, it contains the rank of MAT, i.e., the order of the submatrix R11. This is the same as the order of the submatrix T11 in the complete orthogonal factorization of MAT.

6) TOL is an optional argument such that 0<=TOL<1. If TOL is not specified, or is outside [0,1[, the calculations to determine the condition number of MAT are not performed and crude tests on R(j,j) are performed to determine the numerical rank of MAT. If TOL is present and is in [0,1[, the calculations to determine the condition number are performed, the effective rank of MAT is determined and the reciprocal of the condition number is returned in TOL.

7) If it is possible that MAT may not be full rank (i.e., certain columns of MAT are linear combinations of other columns), then the linearly dependent columns can usually be eliminated by using KRANK=0 and TOL=relative precision of the elements in MAT and B. If each element is correct to, say, 5 digits then TOL=0.00001 should be used. Also, it may be helpful to scale the columns of MAT so that all elements are about the same order of magnitude.

For further details on linear least square problems and algorithms to solve them, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Hansen, P.C., Pereyra, V., and Scherer, G., 2012:** Least Squares Data Fitting with Applications. Johns Hopkins University Press, 328 pp.

6.8.10 subroutine llsq_qr_solve2 (mat, b, x, rnorm, comp_resid, krank, tol, min_norm, diagr, beta, ip, tau)

Purpose

LLSQ_QR_SOLVE2 computes solutions to real linear least squares problems of the form:

Minimize 2-norm|| B - MAT * X ||

using an orthogonal factorization with columns pivoting of MAT. MAT is a m-by-n matrix which may be rank-deficient. m>=n or n>m is permitted.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m-by-nb right hand side matrix B and the n-by-nb solution matrix X, respectively.

MAT and B are overwritten with information generated by LLSQ_QR_SOLVE2. The (complete) orthogonal factorization of MAT is saved in arguments MAT, DIAGR, BETA, IP and TAU on output.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n coefficient matrix MAT.

On exit, MAT has been overwritten by details of its (complete) orthogonal factorization. Other parts of the factorization can be obtained if the optional arguments DIAGR, BETA, IP and TAU are present.

See Further Details.

B (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-nb right hand side matrix B.

On exit, if COMP_RESID is present and is equal true, the residual matrix B - MAT * X overwrites B on output.

The shape of B must verify:

- size(B, 1) = size(MAT, 1) = m
- size(B, 2) = size(X, 2) = nb .

X (OUTPUT) real(stnd), dimension(:,:) On exit, the n-by-nb solution matrix X.

The shape of X must verify:

- size(X, 1) = size(MAT, 2) = n
- size(X, 2) = size(B, 2) = nb .

RNORM (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the 2-norm of the residual vectors for the solutions stored columnwise in the matrix X.

The size of RNORM must verify:

- size(RNORM) = size(X, 2) = size(B, 2) = nb .

COMP_RESID (INPUT, OPTIONAL) logical(lgl) On entry, if COMP_RESID is present and is equal true, the residual matrix B - MAT * X overwrites B on exit.

KRANK (INPUT/OUTPUT, OPTIONAL) integer(i4b) On entry, KRANK=k implies that the first k columns of MAT are to be forced into the basis, pivoting is performed on the last n-k columns of MAT.

When KRANK >=min(m,n) is used, pivoting is not performed. This is appropriate when MAT is known to be full rank.

If KRANK is absent or is <=0, pivoting is done on all columns of MAT. This is appropriate if MAT may not be of full rank (i.e. certain columns of MAT are linear combinations of other columns).

On exit, KRANK contains the effective rank of MAT, i.e., the number of independent columns in matrix MAT.

TOL (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, TOL is used to determine the effective rank of MAT, which is then defined as the order of the largest leading triangular submatrix R11 in the QR factorization (with pivoting) of MAT whose estimated condition number, in the 1-norm, is less than 1/TOL. TOL must be set to the relative precision of the elements in MAT and B. If each element is correct to, say, 5 digits then TOL=0.00001 should be used.

TOL must not be greater or equal to 1 or less than 0, otherwise the numerical rank of MAT is determined and the calculations to determine the condition number are not performed. If TOL=0, the numerical rank of MAT is determined, but the condition number is calculated.

On exit, if a condition number is calculated, its reciprocal is returned in TOL. Otherwise, TOL is not changed.

If TOL is absent, the numerical rank of MAT is used and is returned in the optional argument KRANK.

MIN_NORM (INPUT, OPTIONAL) logical(lgl) On entry, if:

- MIN_NORM=true, a complete orthogonal factorization of MAT and the minimun 2-norm solutions are computed.
- MIN_NORM=false or if MIN_NORM is absent, a QR factorization with column pivoting of MAT and solutions are computed such that if the j-th column of MAT is omitted from the basis, X[j,:] is set to zero.

DIAGR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the diagonal elements of the matrix R if MIN_NORM=false or is absent, or the diagonal elements of the matrix T11 if MIN_NORM is present and is equal to true. The diagonal elements of T11 are stored in DIAGR(1:KRANK).

See Further Details.

The size of DIAGR must be min(size(MAT,1) , size(MAT,2)).

BETA (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the scalars factors of the elementary reflectors defining Q.

See Further Details.

The size of BETA must be min(size(MAT,1) , size(MAT,2)).

IP (OUTPUT, OPTIONAL) integer(i4b), dimension(:) On exit, if IP(j)=k, then the j-th column of MAT * P was the k-th column of MAT.

See Further Details.

The size of IP must be size(MAT,2) = n.

TAU (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the scalars factors of the elementary reflectors defining Z in the complete orthogonal factorization of MAT if MIN_NORM is present and is equal to true. Otherwise, TAU is set to 0.

See Further Details.

The size of TAU must be min(size(MAT,1) , size(MAT,2)).

Further Details

1) The routine first computes a QR factorization with (partial) column pivoting on option (see below):

   MAT * P = Q * R

   , here P is a n-by-n permutation matrix, R is an upper triangular or trapezoidal (if n>m) matrix and Q is a m-by-m orthogonal matrix.

   The matrix Q is represented as a product of elementary reflectors

   Q = H(1) * H(2) * . . . * H(k), where k = min(size(MAT,1) , size(MAT,2))

   Each H(i) has the form

   H(i) = I + beta * (v * v') ,

   where beta is a real scalar and v is a real m-element vector with v(1:i-1) = 0. v(i:m) is stored on exit in MAT(i:m,i) and beta in BETA(i).

   The matrix P is represented in the array IP as follows: If IP(j) = i then the jth column of P is the ith canonical unit vector.

   The elements above the diagonal of the array MAT contain the corresponding elements of the triangular matrix R. The elements on the diagonal of R are stored in the array DIAGR.

   R can then be partioned by defining R11 as the largest leading submatrix of R whose estimated condition number, in the 1-norm, is less than 1/TOL (or such that abs(R[j,j])>0 if TOL is absent). The order of R11, KRANK, is the effective rank of MAT.

   This leads to the following partition of R:

   [R11 R12]

   [R21 R22]

   where R21 is zero by construction (since R is an upper triangular or trapezoidal) and R22 is considered to be negligible.

2) If MIN_NORM is present and has the value true, R12 is annihilated by orthogonal transformations from the right, arriving at the complete orthogonal factorization:

   MAT * P = Q * T * Z

   , where Z is a n-by-n orthogonal matrix and T has the form:

   [T11 T12]

   [T21 T22]

   , here T21 (=R21), T12 and T22 (=R22) are zero and T11 is a KRANK-by-KRANK upper triangular matrix.

   The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), which is used to introduce zeros into the kth row of R, is given in the form

   [I 0]

   [0 T(k)]

   where

   T(k) = I + tau * (u(k) * u(k)') and u(k)' = (1 0 z(k))

   tau is a scalar, u(k) is a n-k+1 vector and z(k) is an (n-KRANK) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of R12.

The Z n-by-n orthogonal matrix is given by

Z = Z(1) * Z(2) * ... * Z(KRANK)

On exit, the scalar tau is returned in the kth element of TAU and the vector u(k) in the kth row of MAT, such that the elements of z(k) are in MAT(k,KRANK+1:n).

On exit, the elements above the diagonal of the array section MAT(1:KRANK,1:KRANK) contain the corresponding elements of the triangular matrix T11. The elements of the diagonal of T11 are stored in the array section DIAGR(1:KRANK). The last part of DIAGR is set to zero. In other words, T11 overwrites R11 and Z overwrites R12 on exit.

The minimum 2-norm solution is then

X = [P * Z'](:,:KRANK) * [inv(T11) * Q1' * B]

where inv(T11) is the inverse of T11, Z' is the transpose of Z and Q1 consists of the first KRANK columns of Q.

3) If MIN_NORM is absent or has the value false, a solution is computed as

X = P(:,:KRANK) * [inv(R11) * Q1' * B]

where inv(R11) is the inverse of R11 and Q1 consists of the first KRANK columns of Q. In this case, if the j-th column of MAT is omitted from the basis, X[j] is set to zero and R is not destroyed in MAT.

4) In both cases:

- The 2-norm of the residual vector for the solution in the j-th column of X is given in RNORM[j] if argument RNORM is present.
- If COMP_RESID=true, The residual matrix B - MAT * X overwrites B on exit.

5) On input, if KRANK is present and KRANK=k, the first k columns of MAT are to be forced into the basis. Pivoting is performed on the last n-k columns of MAT.

When KRANK is present and KRANK>=min(m,n) is used, pivoting is not performed. This is appropriate when MAT is known to be full rank.

If KRANK is absent or is present with KRANK<=0, pivoting is done on all columns of MAT.

On output, if KRANK is present, it contains the rank of MAT, i.e., the order of the submatrix R11. This is the same as the order of the submatrix T11 in the complete orthogonal factorization of MAT.

6) TOL is an optional argument such that 0<=TOL<1. If TOL is not specified, or is outside [0,1[, the calculations to determine the condition number of MAT are not performed and crude tests on R(j,j) are performed to determine the numerical rank of MAT. If TOL is present and is in [0,1[, the calculations to determine the condition number are performed, the effective rank of MAT is determined and the reciprocal of the condition number is returned in TOL.

7) If it is possible that MAT may not be full rank (i.e., certain columns of MAT are linear combinations of other columns), then the linearly dependent columns can usually be eliminated by using KRANK=0 and TOL=relative precision of the elements in MAT and B. If each element is correct to, say, 5 digits then TOL=0.00001 should be used. Also, it may be helpful to scale the columns of MAT so that all elements are about the same order of magnitude.

For further details on linear least square problems and algorithms to solve them, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Hansen, P.C., Pereyra, V., and Scherer, G., 2012:** Least Squares Data Fitting with Applications. Johns Hopkins University Press, 328 pp.

6.8.11 subroutine llsq_qr_solve2 (vec, b, x, rnorm, comp_resid, diagr, beta)

Purpose

LLSQ_QR_SOLVE2 computes a solution to a real linear least squares problem:

Minimize 2-norm|| B - VEC * X ||

VEC is a m-element vector, B is a m-element right hand side vector and X is a real scalar.

VEC and B are overwritten with information generated by LLSQ_QR_SOLVE2.

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the m-element coefficient vector VEC.

On exit, VEC contains the vector v of the Householder reflector H.

See Further Details.

B (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the m-element right hand side vector B.

On exit, if COMP_RESID is present and is equal true, the residual vector B - VEC * X overwrites B on output.

The shape of B must verify:

- size(B) = size(VEC) = m .

X (OUTPUT) real(stnd) On exit, the real solution X.

RNORM (OUTPUT, OPTIONAL) real(stnd) On exit, the 2-norm of the residual vector for the solution scalar X.

COMP_RESID (INPUT, OPTIONAL) logical(lgl) On entry, if COMP_RESID is present and is equal true, the residual vector B - VEC * X overwrites B on exit.

DIAGR (OUTPUT, OPTIONAL) real(stnd) On exit, the scalar DIAGR.

See Further Details.

BETA (OUTPUT, OPTIONAL) real(stnd) On exit, the scalar factor BETA of the elementary reflector defining H.

See Further Details.

Further Details

1) The routine first generates a real elementary reflector H of order m, such that

H * VEC = D , with H' * H = I and D' = (DIAGR 0)

where DIAGR is scalar. H is represented in the form

H = I + BETA * (v * v') ,

where BETA is a real scalar and v is a real m-element vector.

2) The solution X is then computed as

   X = [H * B](1) / DIAGR

3) The 2-norm of the residual vector for the solution X can be obtained through the optional argument RNORM as

   2-norm|| [H * B](2:) ||

4) If COMP_RESID=true, The residual vector B - VEC * X overwrites B on exit.

6.8.12 subroutine llsq_qr_solve2 (vec, b, x, rnorm, comp_resid, diagr, beta)

Purpose

LLSQ_QR_SOLVE2 computes solutions to real linear least squares problems of the form:

Minimize 2-norm|| B - VEC * X ||

here VEC is a m-element vector and several right hand side vectors b and solution scalars x can be handled in a single call; they are stored as the columns of the m-by-nb right hand side matrix B and the nb-element solution vector X, respectively.

VEC and B are overwritten with information generated by LLSQ_QR_SOLVE2.

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the m-element coefficient vector VEC.

On exit, VEC contains the vector v of the Householder reflector H.

See Further Details.

B (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-nb right hand side matrix B.

On exit, if COMP_RESID is present and is equal true, the residual matrix B - VEC * X overwrites B on output.

The shape of B must verify:

- size(B, 1) = size(VEC) = m
- size(B, 2) = size(X) = nb .

X (OUTPUT) real(stnd), dimension(:) On exit, the nb-element solution vector X.

The shape of X must verify:

- size(X) = size(B, 2) = nb .

RNORM (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the 2-norms of the residual vectors for the solutions stored in the vector X.

The size of RNORM must verify:

- size(RNORM) = size(X) = size(B, 2) = nb .

COMP_RESID (INPUT, OPTIONAL) logical(lgl) On entry, if COMP_RESID is present and is equal true, the residual matrix B - VEC * X overwrites B on exit.

DIAGR (OUTPUT, OPTIONAL) real(stnd) On exit, the scalar DIAGR.

See Further Details.

BETA (OUTPUT, OPTIONAL) real(stnd) On exit, the scalar factor BETA of the elementary reflector defining H.

See Further Details.

Further Details

1) The routine first generates a real elementary reflector H of order m, such that

   H * VEC = D , with H' * H = I and D' = (DIAGR 0)

   where DIAGR is scalar. H is represented in the form

   H = I + BETA * (v * v') ,

   where BETA is a real scalar and v is a real m-element vector.

2) The solution vector X is then computed as

   X(:) = [H * B](1,:) / DIAGR

3) The 2-norm of the residual vector for the solution X[j] is given in RNORM[j] if argument RNORM is present.

4) If COMP_RESID=true, The residual matrix B - VEC * X overwrites B on exit.

6.8.13 subroutine qr_solve (mat, diagr, beta, b, x, rnorm, comp_resid)

Purpose

QR_SOLVE solves overdetermined or underdetermined real linear systems

MAT * X = B

with a m-by-n matrix MAT, using a QR factorization of MAT as computed by QR_CMP. m>=n or n>m is permitted, but it is assumed that MAT has full rank. B is a m-element right hand side vector and X is a n-element solution vector.

It is assumed that QR_CMP has been used to compute the QR factorization of MAT before QR_SOLVE.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the QR factorization of the real coefficient matrix MAT as returned by QR_CMP.

The elements above the diagonal of the array contain the corresponding elements of R. The elements on and below the diagonal, with the array BETA, represent the orthogonal matrix Q in the QR decomposition of MAT, as a product of elementary reflectors, as returned by QR_CMP.

DIAGR (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the matrix R in the QR decomposition of MAT.

The size of DIAGR must be min(size(MAT,1) , size(MAT,2)).

BETA (INPUT) real(stnd), dimension(:) On entry, the scalars factors of the elementary reflectors defining Q, as returned by QR_CMP.

The size of BETA must be min(size(MAT,1) , size(MAT,2)),

B (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the m-element right hand side vector B.

On exit, if COMP_RESID is present and is equal true, the residual vector B - MAT * X overwrites B.

The size of B must verify:

- size(B) = size(MAT, 1) = m .

X (OUTPUT) real(stnd), dimension(:) On exit, the n-element solution vector X.

The size of X must verify:

size(X) = size(MAT, 2) = n .

RNORM (OUTPUT, OPTIONAL) real(stnd) On exit, the 2-norm of the residual vector for the solution vector X.

COMP_RESID (INPUT, OPTIONAL) logical(lgl) On entry, if COMP_RESID is present and is equal true, the residual vector B - MAT * X overwrites B on exit.

Further Details

1) It is assumed that QR_CMP has been used to compute the QR factorization of MAT before calling QR_SOLVE.
2) If m>=n: the subroutine finds the least squares solution of an overdetermined system, i.e., solves the least squares problem

   Minimize 2-norm|| B - MAT * X ||

   If m<n: the subroutine finds a solution of an underdetermined system

   MAT * X = B
3) The 2-norm of the residual vector for the solution X can be obtained through the optional argument RNORM .
4) If COMP_RESID=true, The m-element residual vector B - MAT * X overwrites B on exit.
5) MAT, DIAGR, BETA are not modified by this routine and can be left in place for successive calls with different right-hand side vectors B.

For further details on linear least square problems and algorithms to solve them, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Hansen, P.C., Pereyra, V., and Scherer, G., 2012:** Least Squares Data Fitting with Applications. Johns Hopkins University Press, 328 pp.

6.8.14 subroutine qr_solve (mat, diagr, beta, b, x, rnorm, comp_resid)

Purpose

QR_SOLVE solves overdetermined or underdetermined real linear systems of the form:

MAT * X = B

with a m-by-n matrix MAT, using a QR factorization of MAT as computed by QR_CMP. m>=n or n>m is permitted, but it is assumed that MAT has full rank.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m-by-nb right hand side matrix B and the n-by-nb solution matrix X, respectively.

It is assumed that QR_CMP has been used to compute the QR factorization of MAT before QR_SOLVE.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the QR factorization of the real coefficient matrix MAT as returned by QR_CMP.

The elements above the diagonal of the array contain the corresponding elements of R. The elements on and below the diagonal, with the array BETA, represent the orthogonal matrix Q in the QR decomposition of MAT, as a product of elementary reflectors, as returned by QR_CMP.

DIAGR (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the matrix R in the QR decomposition of MAT.

The size of DIAGR must be min(size(MAT,1) , size(MAT,2)).

BETA (INPUT) real(stnd), dimension(:) On entry, the scalars factors of the elementary reflectors defining Q, as returned by QR_CMP.

The size of BETA must be min(size(MAT,1) , size(MAT,2)),

B (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-nb right hand side matrix B.

On exit, if COMP_RESID is present and is equal true, the residual matrix B - MAT * X overwrites B on output.

The shape of B must verify:

- size(B, 1) = size(MAT, 1) = m
- size(B, 2) = size(X, 2) = nb .

X (OUTPUT) real(stnd), dimension(:,:) On exit, the n-by-nb solution matrix X.

The shape of X must verify:

- size(X, 1) = size(MAT, 2) = n
- size(X, 2) = size(B, 2) = nb .

RNORM (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the 2-norm of the residual vectors for the solutions stored columnwise in the matrix X.

The size of RNORM must verify:

- size(RNORM) = size(X, 2) = size(B, 2) = nb .

COMP_RESID (INPUT, OPTIONAL) logical(lgl) On entry, if COMP_RESID is present and is equal true, the residual matrix B - MAT * X overwrites B on exit.

Further Details

1) It is assumed that QR_CMP has been used to compute the QR factorization of MAT before calling QR_SOLVE.
2) If m>=n: the subroutine finds the least squares solutions of overdetermined systems, i.e., solves least squares problems of the form

Minimize 2-norm|| B - MAT * X ||

If m<n: the subroutine finds solutions of underdetermined systems of the form

MAT * X = B

In both cases, several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m-by-nb right hand side matrix B and the n-by-nb solution matrix X, respectively.

3) The 2-norm of the residual vector for the solution in the j-th column of X is given in RNORM[j] if argument RNORM is present.

4) If COMP_RESID=true, The residual matrix B - MAT * X overwrites B on exit.

5) MAT, DIAGR, BETA are not modified by this routine and can be left in place for successive calls with different right-hand side matrices B.

For further details on linear least square problems and algorithms to solve them, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Hansen, P.C., Pereyra, V., and Scherer, G., 2012:** Least Squares Data Fitting with Applications. Johns Hopkins University Press, 328 pp.

6.8.15 subroutine qr_solve2 (mat, diagr, beta, ip, krank, b, x, rnorm, comp_resid, tau)

Purpose

QR_SOLVE2 solves overdetermined or underdetermined real linear systems

MAT * X = B

with a m-by-n matrix MAT, using a QR or (complete) orthogonal factorization of MAT as computed by QR_CMP2, PARTIAL_QR_CMP, PARTIAL_RQR_CMP and PARTIAL_RQR_CMP2 subroutines. m>=n or n>m is permitted and MAT may be rank-deficient. B is a m-element right hand side vector and X is a n-element solution vector.

It is assumed that QR_CMP2, PARTIAL_QR_CMP, PARTIAL_RQR_CMP or PARTIAL_RQR_CMP2 have been used to compute the (complete) orthogonal factorization of MAT before calling QR_SOLVE2.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, details of the QR or (complete) orthogonal factorization of the real coefficient matrix MAT as returned by QR_CMP2, PARTIAL_QR_CMP, PARTIAL_RQR_CMP or PARTIAL_RQR_CMP2 subroutines.

See description of QR_CMP2 subroutine for further details.

DIAGR (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the matrix R in the QR decomposition with column pivoting of MAT if TAU is absent or the diagonal elements of the matrix T11 in the complete orthogonal factorization of MAT if TAU is present, as computed by QR_CMP2. If a complete orthogonal factorization has been computed, the diagonal elements of T11 are stored in DIAGR(1:KRANK).

See description of QR_CMP2 subroutine for further details.

The size of DIAGR must be min(size(MAT,1) , size(MAT,2)).

BETA (INPUT) real(stnd), dimension(:) On entry, the scalars factors of the elementary reflectors defining Q in the QR or orthogonal factorization of MAT, as returned by QR_CMP2, PARTIAL_QR_CMP, PARTIAL_RQR_CMP or PARTIAL_RQR_CMP2 subroutines.

See description of QR_CMP2 subroutine for further details.

The size of BETA must be min(size(MAT,1) , size(MAT,2)),

IP (INPUT) integer(i4b), dimension(:) On entry, the permutation P in the QR or (complete) orthogonal factorization of MAT, as returned by QR_CMP2, PARTIAL_QR_CMP, PARTIAL_RQR_CMP or PARTIAL_RQR_CMP2 subroutines.

See description of QR_CMP2 subroutine for further details.

The size of IP must be size(MAT, 2) = n.

KRANK (INPUT) integer(i4b) On entry, KRANK contains the effective rank of MAT, as returned by QR_CMP2, PARTIAL_QR_CMP, PARTIAL_RQR_CMP or PARTIAL_RQR_CMP2 subroutines.

See description of QR_CMP2 subroutine for further details.

B (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the m-element right hand side vector B .

On exit, if COMP_RESID is present and is equal to true, the residual vector B - MAT * X overwrites B .

The size of B must verify:

- size(B) = size(MAT, 1) = m .

X (OUTPUT) real(stnd), dimension(:) On exit, the n-element solution vector X.

The size of X must verify:

- size(X) = size(MAT, 2) = n .

RNORM (OUTPUT, OPTIONAL) real(stnd) On exit, the 2-norm of the residual vector for the solution vector X.

COMP_RESID (INPUT, OPTIONAL) logical(lgl) On entry, if COMP_RESID is present and is equal to true, the residual vector B - MAT * X overwrites B on exit.

TAU (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if TAU is present, a complete orthogonal factorization of MAT has been computed by QR_CMP2 and TAU contains the scalars factors of the elementary reflectors defining Z in this decomposition. Otherwise, only a QR factorization with column pivoting of MAT has been computed by QR_CMP2, PARTIAL_QR_CMP, PARTIAL_RQR_CMP or PARTIAL_RQR_CMP2 subroutines.

See description of QR_CMP2 subroutine for further details.

The size of TAU must be min(size(MAT,1) , size(MAT,2)).

Further Details

1) It is assumed that QR_CMP2, PARTIAL_QR_CMP, PARTIAL_RQR_CMP or PARTIAL_RQR_CMP2 has been used to compute the (complete) orthogonal factorization (if TAU is present) or the QR factorization with column pivoting (if TAU is absent) of MAT before calling QR_SOLVE2.

2) If m>=n: the subroutine finds the least squares solution of an overdetermined system, i.e., solves the least squares problem

Minimize 2-norm|| B - MAT * X ||

If m<n: the subroutine finds a solution of an underdetermined system

MAT * X = B

In both cases, the minimun 2-norm solution is computed if TAU is present. Otherwise, a solution is computed such that if the j-th column of MAT is omitted from the basis, X[j] is set to zero.

3) The 2-norm of the residual vector for the solution X can be obtained through the optional argument RNORM .

4) If COMP_RESID=true, The m-element residual vector B - MAT * X overwrites B on exit.

5) MAT, DIAGR, BETA, IP, KRANK and TAU are not modified by this routine and can be left in place for successive calls with different right-hand side vectors B.

For further details on linear least square problems and algorithms to solve them, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Hansen, P.C., Pereyra, V., and Scherer, G., 2012:** Least Squares Data Fitting with Applications. Johns Hopkins University Press, 328 pp.

6.8.16 subroutine qr_solve2 (mat, diagr, beta, ip, krank, b, x, rnorm, comp_resid, tau)

Purpose

QR_SOLVE2 solves overdetermined or underdetermined real linear systems of the form:

MAT * X = B

with a m-by-n matrix MAT, using a QR or (complete) orthogonal factorization of MAT as computed by QR_CMP2, PARTIAL_QR_CMP, PARTIAL_RQR_CMP and PARTIAL_RQR_CMP2 subroutines. m>=n or n>m is permitted and MAT may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m-by-nb right hand side matrix B and the n-by-nb solution matrix X, respectively.

It is assumed that QR_CMP2, PARTIAL_QR_CMP, PARTIAL_RQR_CMP or PARTIAL_RQR_CMP2 have been used to compute the (complete) orthogonal factorization of MAT before calling QR_SOLVE2.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, details of the QR or (complete) orthogonal factorization of the real coefficient matrix MAT as returned by QR_CMP2, PARTIAL_QR_CMP, PARTIAL_RQR_CMP or PARTIAL_RQR_CMP2 subroutines.

See description of QR_CMP2 subroutine for further details.

DIAGR (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the matrix R in the QR decomposition with column pivoting of MAT if TAU is absent or the diagonal elements of the matrix T11 in the complete orthogonal factorization of MAT if TAU is present, as computed by QR_CMP2. If a complete orthogonal factorization has been computed, the diagonal elements of T11 are stored in DIAGR(1:KRANK).

See description of QR_CMP2 subroutine for further details.

The size of DIAGR must be min(size(MAT,1) , size(MAT,2)).

BETA (INPUT) real(stnd), dimension(:) On entry, the scalars factors of the elementary reflectors defining Q in the QR or orthogonal factorization of MAT, as returned by QR_CMP2, PARTIAL_QR_CMP, PARTIAL_RQR_CMP or PARTIAL_RQR_CMP2 subroutines.

See description of QR_CMP2 subroutine for further details.

The size of BETA must be min(size(MAT,1) , size(MAT,2)),

IP (INPUT) integer(i4b), dimension(:) On entry, the permutation P in the QR or (complete) orthogonal factorization of MAT, as returned by QR_CMP2, PARTIAL_QR_CMP, PARTIAL_RQR_CMP or PARTIAL_RQR_CMP2 subroutines.

See description of QR_CMP2 subroutine for further details.

The size of IP must be size(MAT,2) = n.

KRANK (INPUT) integer(i4b) On entry, KRANK contains the effective rank of MAT, as returned by QR_CMP2, PARTIAL_QR_CMP, PARTIAL_RQR_CMP or PARTIAL_RQR_CMP2 subroutines.

See description of QR_CMP2 subroutine for further details.

B (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-nb right hand side matrix B .

On exit, if COMP_RESID is present and is equal to true, the residual matrix B - MAT * X overwrites B .

The shape of B must verify:

- size(B, 1) = size(MAT, 1) = m
- size(B, 2) = size(X, 2) = nb .

X (OUTPUT) real(stnd), dimension(:,:) On exit, the n-by-nb solution matrix X.

The shape of X must verify:

- size(X, 1) = size(MAT, 2) = n
- size(X, 2) = size(B, 2) = nb .

RNORM (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the 2-norm of the residual vectors for the solutions stored columnwise in the matrix X.

The size of RNORM must verify:

- size(RNORM) = size(X, 2) = size(B, 2) = nb .

COMP_RESID (INPUT, OPTIONAL) logical(lgl) On entry, if COMP_RESID is present and is equal to true, the residual matrix B - MAT * X overwrites B on exit.

TAU (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if TAU is present, a complete orthogonal factorization of MAT has been computed by QR_CMP2 and TAU contains the scalars factors of the elementary reflectors defining Z in this decomposition. Otherwise, only a QR factorization with column pivoting of MAT has been computed by QR_CMP2, PARTIAL_QR_CMP, PARTIAL_RQR_CMP or PARTIAL_RQR_CMP2 subroutines.

See description of QR_CMP2 subroutine further details.

The size of TAU must be min(size(MAT,1) , size(MAT,2)).

Further Details

1) It is assumed that QR_CMP2, PARTIAL_QR_CMP, PARTIAL_RQR_CMP or PARTIAL_RQR_CMP2 has been used to compute the (complete) orthogonal factorization (if TAU is present) or the QR factorization with column pivoting (if TAU is absent) of MAT before calling QR_SOLVE2.

2) If m>=n: the subroutine finds the least squares solutions of overdetermined systems, i.e., solves least squares problems of the form

   Minimize 2-norm|| B - MAT * X ||

   If m<n: the subroutine finds solutions of underdetermined systems of the form

   MAT * X = B

   In both cases, several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m-by-nb right hand side matrix B and the n-by-nb solution matrix X, respectively.

   In both cases, the minimun 2-norm solutions are computed if TAU is present. Otherwise, solutions are computed such that if the j-th column of MAT is omitted from the basis, X[j,:] is set to zero.

3) The 2-norm of the residual vector for the solution in the j-th column of X is given in RNORM[j] if argument RNORM is present.

4) If COMP_RESID=true, The residual matrix B - MAT * X overwrites B on exit.

5) MAT, DIAGR, BETA, IP, KRANK and TAU are not modified by this routine and can be left in place for successive calls with different right-hand side matrices B.

For further details on linear least square problems and algorithms to solve them, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Hansen, P.C., Pereyra, V., and Scherer, G., 2012:** Least Squares Data Fitting with Applications. Johns Hopkins University Press, 328 pp.

6.8.17 `subroutine rqb_solve ( q, b, c, x, ip, tau, comp_resid )`

Purpose

RQB_SOLVE solves overdetermined or underdetermined real linear systems

MAT * X = (Q * B) * X = C

with a m-by-n matrix MAT, using a randomized QR or complete orthogonal factorization of MAT as computed by RQB_CMP. m>=n or n>m is permitted. Q is a m-by-nqb matrix with orthonormal columns and B is a nqb-by-n upper trapezoidal matrix as computed by RQB_CMP with argument COMP_QR equals to true or with optional arguments IP and/or TAU present. C is a m-element right hand side vector and X is an approximate n-element solution vector.

It is assumed that RQB_CMP has been used to compute the randomized QR or complete orthogonal factorization of MAT before calling RQB_SOLVE.

Arguments

Q (INPUT) real(stnd), dimension(:,:) On entry, the computed m-by-nqb orthonormal matrix of the randomized partial QR or complete orthogonal factorization of MAT as computed by RQB_CMP with COMP_QR equals to true or with optional arguments IP and/or TAU present.

See Further Details.

The shape of Q must verify:

- size(Q, 1) = m ,
- size(Q, 2) = nqb .

B (INPUT) real(stnd), dimension(:,:) On entry, the upper trapezoidal matrix R (or T) of the randomized partial QR (or complete orthogonal) factorization of MAT as computed by RQB_CMP with COMP_QR equals to true or with optional arguments IP and/or TAU present.

See Further Details.

The shape of B must verify:

- size(B, 1) = size(Q, 2) = nqb ,
- size(B, 2) = size(X) = n .

C (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the m-element right hand side vector C.

On exit, if COMP_RESID is present and is equal true, the approximate residual vector C - MAT * X overwrites C.

The size of C must verify:

- size(C) = size(Q, 1) = m .

X (OUTPUT) real(stnd), dimension(:) On exit, the approximate n-element solution vector X.

The size of X must verify:

- size(X) = size(B, 2) = n .

IP (INPUT, OPTIONAL) integer(i4b), dimension(:) On entry, if IP is present a randomized partial QR or complete orthogonal factorization with column pivoting of MAT has been computed by RQB_CMP (e.g., the IP argument has also been specified in the call of RQB_CMP). If IP(j)=k, then the j-th column of MAT*P was the k-th column of MAT.

If IP is present, RQB_SOLVE will reorder the elements of the solution vector X to compensate for the interchanges performed in the column pivoting phase of the QR or orthogonal factorization as computed by RQB_CMP.

See Further Details.

The size of IP must verify:

- size(IP) = size(X) = size(B, 2) = n.

TAU (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if TAU is present, a randomized partial complete orthogonal factorization of MAT has been computed by RQB_CMP (e.g., the TAU argument has also been specified in the call of RQB_CMP) and TAU stores the scalars factors of the elementary reflectors defining Z in the orthogonal factorization of MAT as computed by RQB_CMP.

If TAU is present, RQB_SOLVE will compute the approximate minimal 2-norm solution vector of the linear least-squares problem with the help of the complete orthogonal factorization of MAT as computed by RQB_CMP.

See Further Details.

The size of TAU must verify:

- size(TAU) = size(Q, 2) = size(B, 1) = nqb.

COMP_RESID (INPUT, OPTIONAL) logical(lgl) On entry, if COMP_RESID is present and is equal true, the residual vector C - MAT * X overwrites C on exit.

Further Details

1) It is assumed that RQB_CMP has been used to compute the randomized partial QR or complete orthogonal factorization of MAT before calling RQB_SOLVE. RQB_CMP must be called with COMP_QR argument equals to true or with optional arguments IP and TAU eventually present. IF IP or TAU have been specified in the call of RQB_CMP, they must be also specified in the call of RQB_SOLVE.

2) If m>=n: the subroutine finds an approximate least squares solution of an overdetermined system, i.e., solves the least squares problem

   Minimize 2-norm|| C - MAT * X ||

   If m<n: the subroutine finds an approximate solution of an underdetermined system

   MAT * X = C

3) If IP is present, RQB_SOLVE will reorder the elements of the solution vector X to compensate for the interchanges performed in the column pivoting phase of the QR or orthogonal factorization as computed by RQB_CMP.

4) If TAU is present, RQB_SOLVE will compute the (approximate) minimal 2-norm solution of the above least squares problems with the help of the complete orthogonal factorization of MAT as computed by RQB_CMP.

5) If COMP_RESID=true, The m-element residual vector C - MAT * X overwrites C on exit.

For further details on linear least square problems and algorithms to solve them, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Hansen, P.C., Pereyra, V., and Scherer, G., 2012:** Least Squares Data Fitting with Applications. Johns Hopkins University Press, 328 pp.

6.8.18 subroutine rqb_solve (q, b, c, x, ip, tau, comp_resid)

Purpose

RQB_SOLVE solves overdetermined or underdetermined real linear systems

MAT * X = (Q * B) * X = C

with a m-by-n matrix MAT, using a randomized QR or complete orthogonal factorization of MAT as computed by RQB_CMP. m>=n or n>m is permitted. Q is a m-by-nqb matrix with orthonormal columns and B is a nqb-by-n upper trapezoidal matrix as computed by RQB_CMP with argument COMP_QR equals to true or with optional arguments IP and/or TAU present. C is a m-by-nc right hand side matrix and X is an approximate n-by-nc solution matrix.

It is assumed that RQB_CMP has been used to compute the randomized QR or complete orthogonal factorization of MAT before calling RQB_SOLVE.

Arguments

Q (INPUT) real(stnd), dimension(:,:) On entry, the computed m-by-nqb orthonormal matrix of the randomized partial QR or complete orthogonal factorization of MAT as computed by RQB_CMP with COMP_QR equals to true or with optional arguments IP and/or TAU present.

See Further Details.

The shape of Q must verify:

- size(Q, 1) = m ,
- size(Q, 2) = nqb .

B (INPUT) real(stnd), dimension(:,:) On entry, the upper trapezoidal matrix R (or T) of the randomized partial QR (or complete orthogonal) factorization of MAT as computed by RQB_CMP with COMP_QR equals to true or with optional arguments IP and/or TAU present.

See Further Details.

The shape of B must verify:

- size(B, 1) = size(Q, 2) = nqb ,
- size(B, 2) = size(X, 1) = n .

C (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-nc right hand side matrix C.

On exit, if COMP_RESID is present and is equal true, the approximate residual matrix C - MAT * X overwrites C.

The shape of C must verify:

- size(C, 1) = size(Q, 1) = m .
- size(C, 2) = size(X, 2) = nc .

X (OUTPUT) real(stnd), dimension(:,:) On exit, the approximate n-by-nc solution matrix X.

The shape of X must verify:

- size(X, 1) = size(B, 2) = n
- size(X, 2) = size(C, 2) = nc .

IP (INPUT, OPTIONAL) integer(i4b), dimension(:) On entry, if IP is present a randomized partial QR or complete orthogonal factorization with column pivoting of MAT has been computed by RQB_CMP (e.g., the IP argument has also been specified in the call of RQB_CMP). If IP(j)=k, then the j-th column of MAT*P was the k-th column of MAT.

If IP is present, RQB_SOLVE will reorder the rows of the solution matrix X to compensate for the interchanges performed in the column pivoting phase of the QR or orthogonal factorization as computed by RQB_CMP.

See Further Details.

The size of IP must verify:

- size(IP) = size(X) = size(B, 2) = n.

TAU (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if TAU is present, a randomized partial complete orthogonal factorization of MAT has been computed by RQB_CMP (e.g., the TAU argument has also been specified in the call of RQB_CMP) and TAU stores the scalars factors of the elementary reflectors defining Z in the orthogonal factorization of MAT as computed by RQB_CMP.

If TAU is present, RQB_SOLVE will compute the approximate minimal 2-norm solution matrix of the linear least-squares problem with the help of the complete orthogonal factorization of MAT as computed by RQB_CMP.

See Further Details.

The size of TAU must verify:

- size(TAU) = size(Q, 2) = size(B, 1) = nqb.

COMP_RESID (INPUT, OPTIONAL) logical(lgl) On entry, if COMP_RESID is present and is equal true, the residual matrix C - MAT * X overwrites C on exit.

Further Details

1) It is assumed that RQB_CMP has been used to compute the randomized partial QR or complete orthogonal factorization of MAT before calling RQB_SOLVE. RQB_CMP must be called with COMP_QR argument equals to true or with optional arguments IP and TAU eventually present. IF IP or TAU have been specified in the call of RQB_CMP, they must be also specified in the call of RQB_SOLVE.

2) If m>=n: the subroutine finds approximate least squares solutions of overdetermined systems, i.e., solves the least squares problem

   Minimize 2-norm|| C - MAT * X ||

   If m<n: the subroutine finds an approximate solution of an underdetermined system

   MAT * X = C

3) If IP is present, RQB_SOLVE will reorder the rows of the solution matrix X to compensate for the interchanges performed in the column pivoting phase of the QR or orthogonal factorization as computed by RQB_CMP.

4) If TAU is present, RQB_SOLVE will compute the (approximate) minimal 2-norm solutions of the above least squares problems with the help of the complete orthogonal factorization of MAT as computed by RQB_CMP.

3) If COMP_RESID=true, The m-by-nc residual MATRIX C - MAT * X overwrites C on exit.

For further details on linear least square problems and algorithms to solve them, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Hansen, P.C., Pereyra, V., and Scherer, G., 2012:** Least Squares Data Fitting with Applications. Johns Hopkins University Press, 328 pp.

6.8.19 subroutine llsq_svd_solve (mat, b, failure, x, singvalues, krank, rnorm, tol, mul_size, maxiter, max_francis_steps, perfect_shift, bisect, dqds)

Purpose

LLSQ_SVD_SOLVE computes the minimum norm solution to a real linear least squares problem:

Minimize 2-norm|| B - MAT * X ||

using the singular value decomposition (SVD) of MAT. MAT is a m-by-n matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m-by-nrhs right hand side matrix B and the n-by-nrhs solution matrix X, respectively.

The effective rank of MAT, KRANK, is determined by treating as zero those singular values which are less than TOL times the largest singular value.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is destroyed. If m>=n, MAT(:n,:n) is overwritten with the right singular vectors of MAT, stored columnwise.

B (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-nrhs right hand side matrix B.

On exit, B is destroyed. If m>KRANK, the residual sum-of-squares for the solution in the i-th column is given by the sum of squares of elements KRANK+1:m in that column.

The shape of B must verify:

- size(B, 1) = size(MAT, 1) = m
- size(B, 2) = size(X, 2) = nrhs .

FAILURE (OUTPUT) logical(lgl) If:

- FAILURE= false : indicates successful exit
- FAILURE= true : indicates that the SVD algorithm did not converge and that full accuracy was not attained in the bidiagonal SVD of an intermediate bidiagonal form BD of MAT.

X (OUTPUT) real(stnd), dimension(:,:) On exit, the n-by-nrhs solution matrix X.

The shape of X must verify:

- size(X, 1) = size(MAT, 2) = n
- size(X, 2) = size(B, 2) = nrhs .

SINGVALUES (OUTPUT, OPTIONAL) real(stnd), dimension(:) The singular values of MAT in decreasing order. The condition number of MAT in the 2-norm is

SINGVALUES(1)/SINGVALUES(min(m,n)).

The size of SINGVALUES must verify: size(SINGVALUES) = min(m,n) .

KRANK (OUTPUT, OPTIONAL) integer(i4b) On exit, the effective rank of MAT, i.e., the number of singular values which are greater than TOL * SINGVALUES(1).

RNORM (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the 2-norms of the residual vectors for the solutions stored columnwise in the matrix X.

The size of RNORM must verify:

- size(RNORM) = size(X, 2) = size(B, 2) = nrhs .

TOL (INPUT, OPTIONAL) real(stnd) On entry, TOL is used to determine the effective rank of MAT. Singular values SINGVALUES(i) <= TOL * SINGVALUES(1) are treated as zero. If TOL is absent, machine precision is used instead.

MUL_SIZE (INPUT, OPTIONAL) integer(i4b) Internal parameter. MUL_SIZE must verify 1 <= MUL_SIZE <= max(m,n), otherwise a default value is used. MUL_SIZE can be increased or decreased to improve the performance of the algorithm used in LLSQ_SVD_SOLVE. Maximum performance will be obtained when a real matrix of size MUL_SIZE * max(m,n) and kind stnd fits in the cache of the processors.

The default value is 32.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm. The bidiagonal SVD algorithm of an intermediate bidiagonal form B of MAT fails to converge if the number of QR sweeps exceeds MAXITER * min(m,n). Convergence usually occurs in about 2 * min(m,n) QR sweeps.

The default is 10.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (eg QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm.

MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of min(m,n) and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition and solution matrix at the expense of a slightly slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

This subroutine is adapted from the routine DGELSS in LAPACK. If OPENMP is used, the algorithm is parallelized.

For further details on using the SVD for solving a least square problem, see the references (1), (2) or (3).

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Hansen, P.C., Pereyra, V., and Scherer, G., 2012:** Least Squares Data Fitting with Applications. Johns Hopkins University Press, 328 pp.

6.8.20 subroutine llsq_svd_solve (mat, b, failure, x, singvalues, krank, rnorm, tol, mul_size, maxiter, max_francis_steps, perfect_shift, bisect, dqds)

Purpose

LLSQ_SVD_SOLVE computes the minimum norm solution to a real linear least squares problem:

Minimize 2-norm|| B - MAT * X ||

using the singular value decomposition (SVD) of MAT. MAT is a m-by-n matrix which may be rank-deficient, B is a m-element right hand side vector and X is a n-element solution vector.

The effective rank of MAT, KRANK, is determined by treating as zero those singular values which are less than TOL times the largest singular value.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is destroyed. If m>=n, MAT(:n,:n) is overwritten with the right singular vectors of MAT, stored columnwise.

B (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the m-element right hand side vector B.

On exit, B is destroyed. If m>KRANK, the residual sum-of-squares for the solution X is given by the sum of squares of elements KRANK+1:m of B .

The size of B must verify: size(B) = size(MAT, 1) = m .

FAILURE (OUTPUT) logical(lgl) If:

- FAILURE = false : indicates successful exit
- FAILURE = true : indicates that the SVD algorithm did not converge and that full accuracy was not attained in the bidiagonal SVD of an intermediate bidiagonal form BD of MAT.

X (OUTPUT) real(stnd), dimension(:) On exit, the n-element solution vector X.

The size of X must verify: size(X) = size(MAT, 2) = n .

SINGVALUES (OUTPUT, OPTIONAL) real(stnd), dimension(:) The singular values of MAT in decreasing order. The condition number of MAT in the 2-norm is

SINGVALUES(1)/SINGVALUES(min(m,n)).

The size of SINGVALUES must verify: size(SINGVALUES) = min(m,n) .

KRANK (OUTPUT, OPTIONAL) integer(i4b) On exit, the effective rank of MAT, i.e., the number of singular values which are greater than TOL * SINGVALUES(1).

RNORM (OUTPUT, OPTIONAL) real(stnd) On exit, the 2-norm of the residual vector for the solution vector X.

TOL (INPUT, OPTIONAL) real(stnd) On entry, TOL is used to determine the effective rank of MAT. Singular values SINGVALUES(i) <= TOL * SINGVALUES(1) are treated as zero. If TOL is absent, machine precision is used instead.

MUL_SIZE (INPUT, OPTIONAL) integer(i4b) Internal parameter. MUL_SIZE must verify 1 <= MUL_SIZE <= max(m,n), otherwise a default value is used. MUL_SIZE can be increased or decreased to improve the performance of the algorithm used in LLSQ_SVD_SOLVE. Maximum performance will be obtained when a real matrix of size MUL_SIZE * max(m,n) and kind stnd fits in the cache of the processors.

The default value is 32.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm. The bidiagonal SVD algorithm of an intermediate bidiagonal form B of MAT fails to converge if the number of QR sweeps exceeds MAXITER * min(m,n). Convergence usually occurs in about 2 * min(m,n) QR sweeps.

The default value is 10.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (eg QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm.

MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of min(m,n) and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition and solution vector at the expense of a slightly slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

This subroutine is adapted from the routine DGELSS in LAPACK software. If OPENMP is used, the algorithm is parallelized.

For further details on using the SVD for solving a least square problem, see the references (1), (2) or (3).

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Hansen, P.C., Pereyra, V., and Scherer, G., 2012:** Least Squares Data Fitting with Applications. Johns Hopkins University Press, 328 pp.

6.8.21 subroutine llsq_svd_solve2 (mat, b, failure, x, singvalues, krank, rnorm, tol, mul_size, maxiter, max_francis_steps, perfect_shift, bisect, dqds)

Purpose

LLSQ_SVD_SOLVE2 computes the minimum norm solution to a real linear least squares problem:

Minimize 2-norm|| B - MAT * X ||

using the one-sided Ralha-Barlow bidiagonal reduction algorithm and the singular value decomposition (SVD) of MAT. MAT is a m-by-n matrix with m >= n, which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m-by-nrhs right hand side matrix B and the n-by-nrhs solution matrix X, respectively.

The effective rank of MAT, KRANK, is determined by treating as zero those singular values which are less than TOL times the largest singular value.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is destroyed.

The shape of MAT must verify:

- size(MAT, 1) >= size(MAT, 2) = n .

B (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-nrhs right hand side matrix B.

On exit, B is destroyed. If m>KRANK, the residual sum-of-squares for the solution in the i-th column is given by the sum of squares of elements KRANK+1:m in that column.

The shape of B must verify:

- size(B, 1) = size(MAT, 1) = m
- size(B, 2) = size(X, 2) = nrhs .

FAILURE (OUTPUT) logical(lgl) If:

- FAILURE= false : indicates successful exit
- FAILURE= true : indicates that the SVD algorithm did not converge and that full accuracy was not attained in the bidiagonal SVD of an intermediate bidiagonal form BD of MAT.

X (OUTPUT) real(stnd), dimension(:,:) On exit, the n-by-nrhs solution matrix X.

The shape of X must verify:

- size(X, 1) = size(MAT, 2) = n
- size(X, 2) = size(B, 2) = nrhs .

SINGVALUES (OUTPUT, OPTIONAL) real(stnd), dimension(:) The singular values of MAT in decreasing order. The condition number of MAT in the 2-norm is

SINGVALUES(1)/SINGVALUES(min(m,n)).

The size of SINGVALUES must verify: size(SINGVALUES) = min(m,n) .

KRANK (OUTPUT, OPTIONAL) integer(i4b) On exit, the effective rank of MAT, i.e., the number of singular values which are greater than TOL * SINGVALUES(1).

RNORM (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the 2-norms of the residual vectors for the solutions stored columnwise in the matrix X.

The size of RNORM must verify:

- size(RNORM) = size(X, 2) = size(B, 2) = nrhs .

TOL (INPUT, OPTIONAL) real(stnd) On entry, TOL is used to determine the effective rank of MAT. Singular values SINGVALUES(i) <= TOL * SINGVALUES(1) are treated as zero. If TOL is absent, machine precision is used instead.

MUL_SIZE (INPUT, OPTIONAL) integer(i4b) Internal parameter. MUL_SIZE must verify 1 <= MUL_SIZE <= max(m,n), otherwise a default value is used. MUL_SIZE can be increased or decreased to improve the performance of the algorithm used in LLSQ_SVD_SOLVE. Maximum performance will be obtained when a real matrix of size MUL_SIZE * max(m,n) and kind stnd fits in the cache of the processors.

The default value is 32.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm. The bidiagonal SVD algorithm of an intermediate bidiagonal form B of MAT fails to converge if the number of QR sweeps exceeds MAXITER * min(m,n). Convergence usually occurs in about 2 * min(m,n) QR sweeps.

The default is 10.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (eg QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm.

MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of min(m,n) and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition and solution matrix at the expense of a slightly slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

This subroutine is adapted from the routine DGELSS in LAPACK. If OPENMP is used, the algorithm is parallelized.

This subroutine is faster than the LLSQ_SVD_SOLVE subroutine for solving linear least square problems with a huge coefficient matrix thanks to the use of the Ralha-Barlow one-sided bidiagonal reduction algorithm, but is restricted to the case where the coefficient matrix MAT has more rows than columns.

For further details on the Ralha-Barlow one-sided bidiagonal reduction algorithm and using the SVD for solving a least square problem, see the references (1), (2), (3), (4) and (5).

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Hansen, P.C., Pereyra, V., and Scherer, G., 2012:** Least Squares Data Fitting with Applications. Johns Hopkins University Press, 328 pp.

(4) **Barlow, J.L., Bosner, N., and Drmac, Z., 2005:** A new stable bidiagonal reduction algorithm. Linear Algebra Appl., No 397, pp. 35-84.

(5) **Bosner, N., and Barlow, J.L., 2007:** Block and Parallel versions of one-sided bidiagonalization. SIAM J. Matrix Anal. Appl., Volume 29, No 3, pp. 927-953.

6.8.22 subroutine llsq_svd_solve2 (mat, b, failure, x, singvalues, krank, rnorm, tol, mul_size, maxiter, max_francis_steps, perfect_shift, bisect, dqds)

Purpose

LLSQ_SVD_SOLVE2 computes the minimum norm solution to a real linear least squares problem:

Minimize 2-norm|| B - MAT * X ||

using the one-sided Ralha-Barlow bidiagonal reduction algorithm and the singular value decomposition (SVD) of MAT. MAT is a m-by-n matrix with m >= n, which may be rank-deficient. B is a m-element right hand side vector and X is a n-element solution vector.

The effective rank of MAT, KRANK, is determined by treating as zero those singular values which are less than TOL times the largest singular value.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is destroyed.

The shape of MAT must verify:

- size(MAT, 1) >= size(MAT, 2) = n .

B (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the m-element right hand side vector B.

On exit, B is destroyed. If m>KRANK, the residual sum-of-squares for the solution X is given by the sum of squares of elements KRANK+1:m of B .

The size of B must verify: size(B) = size(MAT, 1) = m .

FAILURE (OUTPUT) logical(lgl) If:

- FAILURE = false : indicates successful exit
- FAILURE = true : indicates that the SVD algorithm did not converge and that full accuracy was not attained in the bidiagonal SVD of an intermediate bidiagonal form BD of MAT.

X (OUTPUT) real(stnd), dimension(:) On exit, the n-element solution vector X.

The size of X must verify: size(X) = size(MAT, 2) = n .

SINGVALUES (OUTPUT, OPTIONAL) real(stnd), dimension(:) The singular values of MAT in decreasing order. The condition number of MAT in the 2-norm is

SINGVALUES(1)/SINGVALUES(min(m,n)).

The size of SINGVALUES must verify: size(SINGVALUES) = min(m,n) .

KRANK (OUTPUT, OPTIONAL) integer(i4b) On exit, the effective rank of MAT, i.e., the number of singular values which are greater than TOL * SINGVALUES(1).

RNORM (OUTPUT, OPTIONAL) real(stnd) On exit, the 2-norm of the residual vector for the solution vector X.

TOL (INPUT, OPTIONAL) real(stnd) On entry, TOL is used to determine the effective rank of MAT. Singular values SINGVALUES(i) <= TOL * SINGVALUES(1) are treated as zero. If TOL is absent, machine precision is used instead.

MUL_SIZE (INPUT, OPTIONAL) integer(i4b) Internal parameter. MUL_SIZE must verify 1 <= MUL_SIZE <= max(m,n), otherwise a default value is used. MUL_SIZE can be increased or decreased to improve the performance of the algorithm used in LLSQ_SVD_SOLVE. Maximum performance will be obtained when a real matrix of size MUL_SIZE * max(m,n) and kind stnd fits in the cache of the processors.

The default value is 32.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm. The bidiagonal SVD algorithm of an intermediate bidiagonal form B of MAT fails to converge if the number of QR sweeps exceeds MAXITER * min(m,n). Convergence usually occurs in about 2 * min(m,n) QR sweeps.

The default value is 10.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (eg QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm.

MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of min(m,n) and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition and solution vector at the expense of a slightly slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

This subroutine is adapted from the routine DGELSS in LAPACK software. If OPENMP is used, the algorithm is parallelized.

This subroutine is faster than the LLSQ_SVD_SOLVE subroutine for solving linear least square problems with a huge coefficient matrix thanks to the use of the Ralha-Barlow one-sided bidiagonal reduction algorithm, but is restricted to the case where the coefficient matrix MAT has more rows than columns.

For further details on the Ralha-Barlow one-sided bidiagonal reduction algorithm and using the SVD for solving a least square problem, see the references (1), (2), (3), (4) and (5).

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Hansen, P.C., Pereyra, V., and Scherer, G., 2012:** Least Squares Data Fitting with Applications. Johns Hopkins University Press, 328 pp.

(4) **Barlow, J.L., Bosner, N., and Drmac, Z., 2005:** A new stable bidiagonal reduction algorithm. Linear Algebra Appl., No 397, pp. 35-84.

(5) **Bosner, N., and Barlow, J.L., 2007:** Block and Parallel versions of one-sided bidiagonalization. SIAM J. Matrix Anal. Appl., Volume 29, No 3, pp. 927-953.

6.9 Module_Lapack_Interfaces

Copyright 2022 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING GENERIC INTERFACES FOR SELECTED SUBROUTINES AND FUNCTIONS IN THE LAPACK LIBRARY.

THIS INTERFACE MODULE ENSURES THAT CALLS TO LAPACK ROUTINES ARE CORRECT, WHEN USED WITH STATPACK.

GENERIC INTERFACES ARE PRESENTLY PROVIDED FOR THE FOLLOWING LAPACK ROUTINES AND DRIVERS :

> Xsytrd, Xorgtr, Xormtr, Xsyev, Xsyevd, Xsyevr, Xsyevx, Xspev, Xspevd, Xspevx, Xsygv, Xsygvd, Xsygvx, Xsteqr, Xstedc, Xstemr, Xstev, Xstevd, Xstevr, Xstevx, Xgeev, Xgeevx, Xgebrd, Xorgbr, Xormbr, Xgesvd, Xgesdd, Xgesvdx, Xbdsqr, Xbdsdc, Xbdsvdx, Xgesv, Xsysv, Xposv, Xgelsd, Xgelss, Xgelsy, Xgels, Xgeqr2, Xgeqrf, Xgeqp3, Xtpqrt, Xtpmqrt

WHERE X CAN BE s, d, c AND z. THE GENERIC INTERFACES HAVE THE FORM:

sytrd, orgtr, ormtr, syev, syevd, syevr, syevx, spev, spevd, spevx, sygv, sygvd, sygvx, steqr, stedc, stemr, stev, stevd, stevr, stevx, geev, geevx, gebrd, orgbr, ormbr, gesvd, gesdd, gesvdx, bdsqr, bdsdc, bdsvdx, gesv, sysv, posv, gelsd, gelss, gelsy, gels, geqr2, geqrf, geqp3, tpqrt, tpmqrt

LATEST REVISION : 19/08/2022

6.10 Module_Lin_Procedures

Copyright 2025 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING SUBROUTINES AND FUNCTIONS FOR THE SOLUTION OF SYSTEMS OF LINEAR EQUATIONS, COMPUTING A TRIANGULAR FACTORIZATION (LU, CHOLESKY), COMPUTING THE INVERSE OF A MATRIX AND COMPUTING THE DETERMINANT OF A MATRIX.

LATEST REVISION : 03/09/2025

6.10.1 `subroutine lu_cmp ( mat, ip, d1, d2, tol, small )`

Purpose

LU_CMP computes the LU decomposition with partial pivoting and implicit row scaling of a given n-by-n real matrix MAT

P * MAT = L * U

where P is a permutation matrix, L is a n-by-n unit lower triangular matrix and U is a n-by-n upper triangular matrix. P is a permutation matrix, stored in argument IP, such that

P = P(n) * ... * P(1)

with P(i) is the identity with row i and IP(i) interchanged.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the coefficient matrix MAT.

On exit, MAT is replaced by the LU decomposition of a rowwise permutation of MAT. The unit diagonal of L is not stored. For solving efficiency, the diagonal reciprocals of the matrix U are saved in the diagonal entries of MAT.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

IP (OUTPUT) integer(i4b), dimension(:) On exit, IP records the permutations effected by the partial pivoting.

The size of IP must verify: size(IP) = n .

D1 (OUTPUT) real(stnd) On exit, if D2 is absent:

- D1 = +1, if an even number of interchanges was carried out
- D1 = -1, if an odd number of interchanges was carried out
- D1 = 0, if MAT is algorithmically singular.

On exit, if D2 is present, D1 is the first component of the determinant of MAT (mantissa of determinant).

D2 (OUTPUT, OPTIONAL) integer(i4b) If D2 is present, the two components of the determinant of MAT are computed.

On exit, D2 is the second component of the determinant of MAT (exponent of determinant):

determinant(MAT) = scale(D1, D2)

TOL (INPUT, OPTIONAL) real(stnd) On entry, a relative tolerance used to indicate whether or not MAT is nearly singular. Tol should normally be choose as approximately the largest relative error in the elements of MAT. For example, if the elements of MAT are correct to about 4 significant figures, then TOL should be set to about 5 * 10**(-4).

If TOL is supplied as less than EPS, where EPS is the relative machine precision, then the value EPS is used in place of TOL.

Default: EPS, the relative machine precision.

SMALL (INPUT, OPTIONAL) real(stnd) On entry, if the system is singular, replaces a diagonal term of the matrix U if it is smaller in magnitude than the value SMALL using the same sign as the diagonal term. An approximate solution based on this replacement can be obtained if no overflow results.

If SMALL is supplied as less than SAFMIN, the smallest number that can be reciprocated safely, then the value SAFMIN is used in place of SMALL.

Default: SAFMIN, the smallest number that can be reciprocated safely.

Further Details

MAT is declared singular if a diagonal element of U is such that

abs(U(j,j)) <= n * norm(MAT(j,:)) * TOL

where norm(MAT(j,:)) denotes the maximum of the absolute values of the jth row of the matrix MAT. In this case, a diagonal element of U is small, indicating that MAT is singular or nearly singular and D1 is set to zero.

If D1/=0 then the linear system MAT * X = B can be solved with subroutines LU_SOLVE or LU_SOLVE2.

If MAT is algorithmically singular (D1=0), the diagonal terms of U smaller in magnitude than the value SMALL have been replaced by SMALL, using the same sign as the diagonal terms and the decomposition has been completed. An approximate solution based on this replacement can be obtained if no overflow results.

A blocked algorithm is used to compute the factorization. Furthermore, the computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2011:** Gaussian elimination. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 3, Issue 3, pp 230-238.

6.10.2 subroutine lu_cmp2 (mat, ip, d1, d2, b, matinv, tol, small)

Purpose

LU_CMP2 computes the LU decomposition with partial pivoting and implicit row scaling of a given n-by-n real matrix MAT

P * MAT = L * U

where P is a permutation matrix, L is a n-by-n unit lower triangular matrix and U is a n-by-n upper triangular matrix. P is a permutation matrix, stored in argument IP, such that

P = P(n) * ... * P(1)

with P(i) is the identity with row i and IP(i) interchanged.

If D2 is present, LU_CMP2 computes the determinant of MAT as

determinant(MAT) = scale(D1, D2)

If B is present, LU_CMP2 solves the system of linear equations

MAT * X = B

using the LU factorization with scaled partial pivoting of MAT. Here B is a n-vector.

If MATINV is present, LU_CMP2 computes the inverse of MAT

MATINV = MAT**(-1)

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the coefficient matrix MAT.

On exit, MAT is replaced by the LU decomposition of a rowwise permutation of MAT. The unit diagonal of L is not stored. For solving efficiency, the diagonal reciprocals of the matrix U are saved in the diagonal entries of MAT.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

IP (OUTPUT) integer(i4b), dimension(:) On exit, IP records the permutations effected by the partial pivoting.

The size of IP must verify: size(IP) = n .

D1 (OUTPUT) real(stnd) On exit, if D2 is absent:

- D1 = +1, if an even number of interchanges was carried out
- D1 = -1, if an odd number of interchanges was carried out
- D1 = 0, if MAT is algorithmically singular.

On exit, if D2 is present, D1 is the first component of the determinant of MAT (mantissa of determinant).

D2 (OUTPUT, OPTIONAL) integer(i4b) If D2 is present, the components of the determinant of MAT are computed.

On exit, D2 is the second component of the determinant of MAT (exponent of determinant) :

determinant(MAT) = scale(D1, D2)

B (INPUT/OUTPUT, OPTIONAL) real(stnd), dimension(:) On entry, the right hand side vector B.

On exit, the solution vector X.

The shape of B must verify: size(B) = n .

MATINV (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, if MAT is not singular, MATINV contains the inverse of MAT.

The shape of MATINV must verify: size(MATINV, 1) = size(MATINV, 2) = n .

TOL (INPUT, OPTIONAL) real(stnd) On entry, a relative tolerance used to indicate whether or not MAT is nearly singular. Tol should normally be choose as approximately the largest relative error in the elements of MAT. For example, if the elements of MAT are correct to about 4 significant figures, then TOL should be set to about 5 * 10**(-4). If TOL is supplied as less than EPS, where EPS is the relative machine precision, then the value EPS is used in place of TOL.

Default: EPS, the relative machine precision.

SMALL (INPUT, OPTIONAL) real(stnd) On entry, if the system is singular, replaces a diagonal term of the matrix U if it is smaller in magnitude than the value SMALL using the same sign as the diagonal term. An approximate solution for X based on this replacement can be obtained if no overflow results. If SMALL is supplied as less than SAFMIN, the smallest number that can be reciprocated safely, then the value SAFMIN is used in place of SMALL.

Default: SAFMIN, the smallest number that can be reciprocated safely.

Further Details

MAT is declared singular if a diagonal element of U is such that

abs(U(j,j)) <= n * norm(MAT(j,:)) * TOL

where norm(MAT(j,:)) denotes the maximum of the absolute values of the jth row of the matrix MAT. In this case, a diagonal element of U is small, indicating that MAT is singular or nearly singular and D1 is set to zero.

If MAT is algorithmically singular (D1=0), the diagonal terms of U smaller in magnitude than the value SMALL have been replaced by SMALL, using the same sign as the diagonal terms, and the decomposition has been completed. An approximate solution for X based on this replacement is then obtained if no overflow results and MATINV is filled with nan() value.

If D1/=0 then the linear system MAT * Z = D can be solved with subroutines LU_SOLVE or LU_SOLVE2.

The computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2011:** Gaussian elimination. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 3, Issue 3, pp 230-238.

6.10.3 `subroutine chol_cmp ( mat, invdiag, d1, d2, upper, tol )`

Purpose

CHOL_CMP computes the Cholesky factorization of a n-by-n real symmetric positive definite matrix MAT. The factorization has the form

MAT = U' * U , if UPPER=true or is absent,

and

MAT = L * L' , if UPPER=false,

where U is an upper triangular matrix and L is a lower triangular matrix.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric positive definite matrix MAT. If:

- UPPER = true or is absent: The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix and the strictly lower triangular part of MAT is not referenced.
- UPPER = false: The leading n-by-n lower triangular part of MAT contains the lower triangular part of the matrix and the strictly upper triangular part of MAT is not referenced.

On exit, if D1/=0, the factor U or L from the Cholesky factorization MAT = U' * U or MAT = L * L', except for the main diagonal elements which are stored in reciprocal form in INVDIAG. The main diagonal elements of MAT are not modified.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

INVDIAG (OUTPUT) real(stnd), dimension(:) On exit, INVDIAG contains the reciprocals of the actual diagonal elements of L or U.

The size of INVDIAG must verify: size(INVDIAG) = n .

D1 (OUTPUT) real(stnd) On exit, D1 = zero indicates that the matrix MAT is algorithmically not positive definite and that the factorization can not be completed. Any other value indicates successful exit.

On exit, if D2 is present, D1 is the first component of the determinant (mantissa of determinant) of MAT.

D2 (OUTPUT, OPTIONAL) integer(i4b) If D2 is present, the components of the determinant of MAT are computed.

On exit, D2 is the second component of the determinant of MAT (exponent of determinant):

determinant(MAT) = scale(D1, D2)

UPPER (INPUT, OPTIONAL) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular is stored
- UPPER = false: Lower triangular is stored.

The default is true.

TOL (INPUT, OPTIONAL) real(stnd) Tolerance used to test the matrix for positive-definiteness. TOL is used as a multiplying factor for determining effective zero for pivots. TOL must be greater or equal to zero, otherwise the default value is used.

The default value is the machine precision multiplied by n.

Further Details

MAT is declared not positive definite if during the j-th stage of the factorization of MAT, a pivot, PIV(j), is such that

PIV(j) <= MAT(j,j) * TOL

In this case, the leading minor of order j of MAT is declared not positive definite and on exit of CHOL_CMP:

- D1 is set to zero,
- INVDIAG(j) = PIV(j),
- INVDIAG(j+1_i4b:n) are set to nan() value,

and the Cholesky factorization is not completed.

On the other hand, if MAT is positive definite then

U(j,j) = sqrt(PIV(j)) (if UPPER=true), for j=1 to n,

or

L(j,j) = sqrt(PIV(j)) (if UPPER=false), for j=1 to n,

and on exit of CHOL_CMP:

- D1/=0,
- INVDIAG(j)=1/sqrt(PIV(j)) for j=1 to n,

and the linear system MAT * Z = D can be solved with subroutine CHOL_SOLVE.

This is a GAxpy version of the Cholesky algorithm, for more details see the reference (1).

A blocked algorithm is used to compute the factorization. Furthermore, the computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2009:** Cholesky factorization. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 1, pp 251-254.

6.10.4 subroutine chol_cmp2 (mat, invdiag, d1, d2, b, matinv, upper, fill, tol)

Purpose

CHOL_CMP2 computes the Cholesky factorization of a n-by-n real symmetric positive definite matrix MAT. The factorization has the form

MAT = U' * U , if UPPER=true or is absent,

and

MAT = L * L' , if UPPER=false,

where U is an upper triangular matrix and L is a lower triangular matrix.

If D2 is present, CHOL_CMP2 computes the determinant of MAT as

determinant(MAT) = scale(D1, D2)

If B is present, CHOL_CMP2 solves the system of linear equations

MAT * X = B

using the Cholesky factorization of MAT. Here B is a n-vector.

If MATINV is present, CHOL_CMP2 computes the inverse of MAT

MATINV = MAT**(-1)

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT. If:

- UPPER = true or is absent: The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix and the strictly lower triangular part of MAT is not referenced.
- UPPER = false: The leading n-by-n lower triangular part of MAT contains the lower triangular part of the matrix and the strictly upper triangular part of MAT is not referenced.

On exit, if D1/=0, the factor U or L from the Cholesky factorization MAT = U' * U or MAT = L * L', except for the main diagonal elements which are stored in reciprocal form in INVDIAG. The main diagonal elements of MAT are not modified.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

INVDIAG (OUTPUT) real(stnd), dimension(:) On exit, INVDIAG contains the reciprocals of the actual diagonal elements of L or U.

The size of INVDIAG must verify: size(INVDIAG) = n .

D1 (OUTPUT) real(stnd) On exit, D1 = zero indicates that the matrix MAT is algorithmically not positive definite and that the factorization can not be completed. Any other value indicates successful exit.

On exit, if D2 is present, D1 is the first component of the determinant (mantissa of determinant) of MAT.

D2 (OUTPUT, OPTIONAL) integer(i4b) If D2 is present, the components of the determinant of MAT are computed.

On exit, D2 is the second component of the determinant of MAT (exponent of determinant)

determinant(MAT) = scale(D1, D2)

B (INPUT/OUTPUT, OPTIONAL) real(stnd), dimension(:) On entry, the right hand side vector B. On exit, the solution vector X.

The shape of B must verify: size(B) = n .

MATINV (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, if:

- FILL = true or is absent: The (symmetric) inverse of MAT.
- FILL = false: The upper (if UPPER=true) or lower (if UPPER=false) triangle of the (symmetric) inverse of MAT, is stored in the upper or lower triangular part of the matrix MATINV and the other part of MATINV is not referenced.

The shape of MATINV must verify: size(MATINV, 1) = size(MATINV, 2) = n .

UPPER (INPUT, OPTIONAL) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular is stored
- UPPER = false: Lower triangular is stored.

The default is true.

FILL (INPUT, OPTIONAL) logical(lgl) On entry, when argument FILL is present, FILL is used as follows. If:

- FILL= true and UPPER= true, the lower triangle of MATINV is filled on exit
- FILL= true and UPPER= false, the upper triangle of MATINV is filled on exit
- FILL= false, the lower (UPPER= true) or upper (UPPER= false) triangle of MATINV is not filled on exit.

The default is true.

TOL (INPUT, OPTIONAL) real(stnd) Tolerance used to test the matrix for positive-definiteness. TOL is used as a multiplying factor for determining effective zero for pivots. TOL must be greater or equal to zero, otherwise the default value is used.

The default value is the machine precision multiplied by n.

Further Details

MAT is declared not positive definite if during the j-th stage of the factorization of MAT, a pivot, PIV(j), is such that

PIV(j) <= MAT(j,j) * TOL

In this case, the leading minor of order j of MAT is declared not positive definite and on exit of CHOL_CMP2:

- D1 is set to zero,
- INVDIAG(j) = PIV(j),
- INVDIAG(j+1_i4b:n) are set to nan() value,
- B is filled with nan() value,
- the upper or lower triangle of MATINV is filled with nan() value if FILL=false,
- the matrix MATINV is filled with nan() value if FILL=true,

and the Cholesky factorization is not completed.

On the other hand, if MAT is positive definite then

U(j,j) = sqrt(PIV(j)) (if UPPER=true), for j=1 to n,

or

L(j,j) = sqrt(PIV(j)) (if UPPER=false), for j=1 to n,

and on exit of CHOL_CMP2:

- D1/=0,
- INVDIAG(j)=1/sqrt(PIV(j)) for j=1 to n,
- B and MATINV are computed, if these arguments are present,

and the linear system MAT * Z = D can be solved with subroutine CHOL_SOLVE.

This is a GAxpy version of the Cholesky algorithm, for more details see the reference (1).

A blocked algorithm is used to compute the factorization. Furthermore, the computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2009:** Cholesky factorization. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 1, pp 251-254.

6.10.5 subroutine gchol_cmp (mat, invdiag, krank, d1, d2, upper, tol)

Purpose

GCHOL_CMP computes the Cholesky factorization of a n-by-n real symmetric positive semidefinite matrix MAT. The factorization has the form

MAT = U' * U , if UPPER=true or is absent,

and

MAT = L * L' , if UPPER=false,

where U is an upper triangular matrix and L is a lower triangular matrix.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT. If:

- UPPER = true or is absent: The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix and the strictly lower triangular part of MAT is not referenced.
- UPPER = false: The leading n-by-n lower triangular part of MAT contains the lower triangular part of the matrix and the strictly upper triangular part of MAT is not referenced.

On exit, if D1>=0, the factor U or L from the Cholesky factorization MAT = U' * U or MAT = L * L', except for the main diagonal elements which are stored in reciprocal form in INVDIAG. The main diagonal elements of MAT are not modified.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

INVDIAG (OUTPUT) real(stnd), dimension(:) On exit, INVDIAG contains the reciprocals of the actual diagonal elements of L or U, excepted for zeroed elements if MAT is not positive definite.

The shape of INVDIAG must verify: size(INVDIAG) = n .

KRANK (OUTPUT) integer(i4b) On exit, KRANK contains the effective rank of MAT, which is defined as the number of nonzero elements of the diagonal of L or U. Note that KRANK may be different from the true rank of MAT. See the reference (2) for details.

D1 (OUTPUT) real(stnd) On exit, D1 < zero indicates that the matrix MAT is algorithmically not positive semidefinite and that the factorization can not be completed. Any other value indicates successful exit.

On exit, if D2 is present, D1 is the first component of the determinant (mantissa of determinant) of MAT.

D2 (OUTPUT, OPTIONAL) integer(i4b) If D2 is present, the components of the determinant of MAT are computed.

On exit, D2 is the second component of the determinant of MAT (exponent of determinant)

determinant(MAT) = scale(D1, D2)

UPPER (INPUT, OPTIONAL) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular is stored
- UPPER = false: Lower triangular is stored.

The default is true.

TOL (INPUT, OPTIONAL) real(stnd) Tolerance used to test MAT for positive-semidefiniteness. TOL is used as a multiplying factor for determining effective zero for pivots. TOL must be greater or equal to zero, otherwise the default value is used.

The default value is the machine precision multiplied by n.

Further Details

MAT is declared not positive semidefinite if during the j-th stage of the factorization of MAT, a pivot, PIV(j), is such that

PIV(j) <= - abs(MAT(j,j) * TOL)

In this case, the leading minor of order j of MAT is declared not positive semidefinite and on exit of GCHOL_CMP:

- D1 is set to -1,
- KRANK is set to -1,
- INVDIAG(j) = PIV(j),
- INVDIAG(j+1_i4b:n) are set to nan() value,

and the Cholesky factorization is not completed.

On the other hand, if MAT is positive semidefinite (e.g. D1>=0), KRANK is computed as follows:

KRANK is initially set to n. if, during the factorization, a pivot, PIV(j), is such that

abs(PIV(j)) <= abs(MAT(j,j) * TOL)

KRANK is decreased by 1 and U(j,j:n) (if UPPER=true) or L(j:n,j) (if UPPER=false) is set to zero. Note that KRANK may be different from the true rank of MAT. See the reference (2) for details.

IF PIV(j) does not satisfy this condition then

- U(j,j) = sqrt(PIV(j)) (if UPPER=true),
- L(j,j) = sqrt(PIV(j)) (if UPPER=false).

On exit of GCHOL_CMP, if MAT is positive semidefinite, INVDIAG contains the reciprocals of the diagonal elements of U or L, excepted for zeroed elements during the factorization as described above.

If MAT is positive semidefinite (D1>=0), the linear system MAT * Z = D can also be solved with subroutine CHOL_SOLVE.

This is a GAxpy version of the Cholesky algorithm, for more details see the reference (1).

A blocked algorithm is used to compute the factorization. Furthermore, the computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2009:** Cholesky factorization. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 1, pp 251-254.

6.10.6 subroutine gchol_cmp2 (mat, invdiag, krank, d1, d2, b, matinv, upper, fill, tol)

Purpose

GCHOL_CMP2 computes the Cholesky factorization of a n-by-n real symmetric positive semidefinite matrix MAT. The factorization has the form

MAT = U' * U , if UPPER=true or is absent,

and

MAT = L * L' , if UPPER=false,

where U is an upper triangular matrix and L is a lower triangular matrix.

If D2 is present, GCHOL_CMP2 computes the determinant of MAT as

determinant(MAT) = scale(D1, D2)

If B is present, GCHOL_CMP2 solves the system of linear equations

MAT * X = B

using the Cholesky factorization of MAT if B belongs to the range of MAT. Here B is a n-vector. IF B does not belongs to the range of MAT, an approximate solution is computed as

X = MATINV * B

where MATINV is a (generalized) inverse of MAT.

If MATINV is present, GCHOL_CMP2 computes a (generalized) inverse of MAT.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix MAT. If:

- UPPER = true or is absent: The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix and the strictly lower triangular part of MAT is not referenced.
- UPPER = false: The leading n-by-n lower triangular part of MAT contains the lower triangular part of the matrix and the strictly upper triangular part of MAT is not referenced.

On exit, if D1>=0, the factor U or L from the Cholesky factorization MAT = U' * U or MAT = L * L', except for the main diagonal elements which are stored in reciprocal form in INVDIAG. The main diagonal elements of MAT are not modified.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

INVDIAG (OUTPUT) real(stnd), dimension(:) On exit, INVDIAG contains the reciprocals of the actual diagonal elements of L or U, excepted for zeroed elements if MAT is not positive definite.

The shape of INVDIAG must verify: size(INVDIAG) = n .

KRANK (OUTPUT) integer(i4b) On exit, KRANK contains the effective rank of MAT, which is defined as the number of nonzero elements of the diagonal of L or U. Note that KRANK may be different from the true rank of MAT. See the reference (2) for details.

D1 (OUTPUT) real(stnd) On exit, D1 < zero indicates that the matrix MAT is algorithmically not positive semidefinite and that the factorization can not be completed. Any other value indicates successful exit.

On exit, if D2 is present, D1 is the first component of the determinant (mantissa of determinant) of MAT.

D2 (OUTPUT, OPTIONAL) integer(i4b) If D2 is present, the components of the determinant of MAT are computed.

On exit, D2 is the second component of the determinant of MAT (exponent of determinant)

determinant(MAT) = scale(D1, D2)

B (INPUT/OUTPUT, OPTIONAL) real(stnd), dimension(:) On entry, the right hand side vector B.

On exit, one solution vector X if B belongs to the range of MAT, otherwise an approximate solution computed with the help of a symmetric generalized inverse of MAT.

The shape of B must verify: size(B) = n .

MATINV (OUTPUT) real(stnd), dimension(:,:) On exit, if:

- FILL = true or is absent: The (symmetric) (generalized) inverse of MAT.
- FILL = false: The upper (if UPPER=true) or lower (if UPPER=false) triangle of the (symmetric) (generalized) inverse of MAT, is stored in the upper or lower triangular part of the matrix MATINV and the other part of MATINV is not referenced.

The shape of MATINV must verify: size(MATINV, 1) = size(MATINV, 2) = n .

UPPER (INPUT, OPTIONAL) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular is stored
- UPPER = false: Lower triangular is stored.

The default is true.

FILL (INPUT, OPTIONAL) logical(lgl) On entry, when argument FILL is present, FILL is used as follows. If:

- FILL= true and UPPER= true, the lower triangle of MATINV is filled on exit.
- FILL= true and UPPER= false, the upper triangle of MATINV is filled on exit.
- FILL= false, the lower (UPPER= true) or upper (UPPER= false) triangle of MATINV is not filled on exit.

The default is true.

TOL (INPUT, OPTIONAL) real(stnd) Tolerance used to test MAT for positive-semidefiniteness. TOL is used as a multiplying factor for determining effective zero for pivots. TOL must be greater or equal to zero, otherwise the default value is used.

The default is the machine precision multiplied by n.

Further Details

MAT is declared not positive semidefinite if during the j-th stage of the factorization of MAT, a pivot, PIV(j), is such that

PIV(j) <= - abs(MAT(j,j) * TOL)

In this case, the leading minor of order j of MAT is declared not positive semidefinite and on exit of GCHOL_CMP2:

- D1 is set to -1,
- KRANK is set to -1,
- INVDIAG(j) = PIV(j),
- INVDIAG(j+1_i4b:n) are set to nan() value,
- B is filled with nan() value,
- the upper or lower triangle of MATINV is filled with nan() value if FILL=false,
- the matrix MATINV is filled with nan() value if FILL=true,

and the Cholesky factorization is not completed.

On the other hand, if MAT is positive semidefinite (D1>=0), KRANK is computed as follows:

KRANK is initially set to n. if, during the factorization, a pivot, PIV(j), is such that

abs(PIV(j)) <= abs(MAT(j,j) * TOL)

KRANK is decreased by 1 and U(j,j:n) (if UPPER=true) or L(j:n,j) (if UPPER=false) is set to zero. Note that KRANK may be different from the true rank of MAT. See the reference (2) for details.

IF PIV(j) does not satisfy this condition then

- U(j,j) = sqrt(PIV(j)) (if UPPER=true),
- L(j,j) = sqrt(PIV(j)) (if UPPER=false).

On exit of GCHOL_CMP2, if MAT is positive semidefinite, INVDIAG contains the reciprocals of the diagonal elements of U or L, excepted for zeroed elements during the factorization as described above.

If MAT is positive semidefinite (D1>=0), MATINV is computed as follows. If:

- KRANK=n, MATINV is just the inverse of MAT, MATINV = MAT**(-1) ,
- KRANK<n, MATINV is a generalized inverse of MAT.

MATINV is a generalized inverse of MAT if

MAT * MATINV * MAT = MAT and MATINV * MAT * MATINV = MATINV

If MAT is positive semidefinite (D1>=0), the linear system MAT * Z = D can also be solved with subroutine CHOL_SOLVE if D belongs to the range of MAT.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2009:** Cholesky factorization. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 1, pp 251-254.

6.10.7 `subroutine lu_solve ( mat, ip, b )`

Purpose

LU_SOLVE solves a system of linear equations

MAT * X = B

where MAT is a n-by-n coefficient matrix and B is a n-vector, using the LU factorization with scaled partial pivoting of MAT, P * MAT = L * U, as computed by LU_CMP or LU_CMP2.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) MAT contains the LU factorization of P * MAT for some permutation matrix P specified by argument IP. It is assumed that MAT is as generated by LU_CMP or LU_CMP2.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

IP (INPUT) integer(i4b), dimension(:) The permutation matrix P as generated by LU_CMP or LU_CMP2.

The shape of IP must verify: size(IP) = n .

B (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the right hand side vector B.

On exit, the solution vector X.

The shape of B must verify: size(B) = n .

Further Details

It is assumed that LU_CMP or LU_CMP2 has been used to compute the LU factorization of MAT before LU_SOLVE.

MAT and IP are not modified by this routine and can be left in place for successive calls with different right-hand sides B.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2011:** Gaussian elimination. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 3, Issue 3, pp 230-238.

6.10.8 subroutine lu_solve (mat, ip, b)

Purpose

LU_SOLVE solves a system of linear equations with several right hand sides

MAT * X = B

where MAT is a n-by-n coefficient matrix and B is a n-by-nb matrix, using the LU factorization with scaled partial pivoting of MAT, P * MAT = L * U, as computed by LU_CMP or LU_CMP2.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) MAT contains the LU factorization of P * MAT for some permutation matrix P specified by argument IP. It is assumed that MAT is as generated by LU_CMP or LU_CMP2.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

IP (INPUT) integer(i4b), dimension(:) The permutation matrix P as generated by LU_CMP or LU_CMP2.

The shape of IP must verify: size(IP) = n .

B (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the right hand side matrix B.

On exit, the solution matrix X.

The shape of B must verify: size(B, 1) = n .

Further Details

It is assumed that LU_CMP or LU_CMP2 has been used to compute the LU factorization of MAT before LU_SOLVE.

MAT and IP are not modified by this routine and can be left in place for successive calls with different right-hand sides B.

The computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.10.9 subroutine lu_solve2 (mat, ip, b)

Purpose

LU_SOLVE2 solves a system of linear equations

MAT * X = B

where MAT is a n-by-n coefficient matrix and B is a n-vector, using the LU factorization with scaled partial pivoting of MAT, P * MAT = L * U, as computed by LU_CMP or LU_CMP2.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) MAT contains the LU factorization of P * MAT for some permutation matrix P specified by argument IP. It is assumed that MAT is as generated by LU_CMP or LU_CMP2.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

IP (INPUT) integer(i4b), dimension(:) The permutation matrix P as generated by LU_CMP or LU_CMP2.

The shape of IP must verify: size(IP) = n .

B (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the right hand side vector B.

On exit, the solution vector X.

The shape of B must verify: size(B) = n .

Further Details

It is assumed that LU_CMP or LU_CMP2 has been used to factor MAT before LU_SOLVE2.

MAT and IP are not modified by this routine and can be left in place for successive calls with different right-hand sides B.

This subroutines takes into account the possibility that B will begin with many zero elements, so it is efficient for use in matrix inversion.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2011:** Gaussian elimination. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 3, Issue 3, pp 230-238.

6.10.10 `subroutine lu_solve2 ( mat, ip, b )`

Purpose

LU_SOLVE2 solves a system of linear equations with several right hand sides

MAT * X = B

where MAT is a n-by-n coefficient matrix and B is a n-by-nb matrix, using the LU factorization with scaled partial pivoting of MAT, P * MAT = L * U, as computed by LU_CMP or LU_CMP2.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) MAT contains the LU factorization of P * MAT for some permutation matrix P specified by argument IP. It is assumed that MAT is as generated by LU_CMP or LU_CMP2.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

IP (INPUT) integer(i4b), dimension(:) The permutation matrix P as generated by LU_CMP or LU_CMP2.

The shape of IP must verify: size(IP) = n .

B (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the right hand side matrix B.

On exit, the solution matrix X.

The shape of B must verify: size(B, 1) = n .

Further Details

It is assumed that LU_CMP or LU_CMP2 has been used to factor MAT before LU_SOLVE2.

MAT and IP are not modified by this routine and can be left in place for successive calls with different right-hand sides B.

This subroutines takes into account the possibility that each column of B will begin with many zero elements, so it is efficient for use in matrix inversion.

The computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.10.11 `function solve_lin ( mat, b, tol )`

Purpose

SOLVE_LIN solves a system of linear equations

MAT * X = B

with a n-by-n coefficient matrix MAT. B is a n-vector.

The function returns the solution vector X, if the matrix MAT is not singular.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) MAT contains the coefficient matrix of the equation

MAT * X = B

MAT is not modified by the subroutine.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

B (INPUT) real(stnd), dimension(:) On entry, the right hand side vector B. B is not modified by the subroutine.

The shape of B must verify: size(B) = n .

TOL (INPUT, OPTIONAL) real(stnd) On entry, a relative tolerance used to indicate whether or not MAT is nearly singular. Tol should normally be choose as approximately the largest relative error in the elements of MAT. For example, if the elements of MAT are correct to about 4 significant figures, then TOL should be set to about 5 * 10**(-4). If TOL is supplied as less than EPS, where EPS is the relative machine precision, then the value EPS is used in place of TOL.

Default: EPS, the relative machine precision.

Further Details

The LU decomposition with partial pivoting and implicit row scaling of the matrix MAT is used to solve the linear system.

MAT is declared singular if a diagonal element of U is such that

abs(U(j,j)) <= n * norm(MAT(j,:)) * TOL

where norm(MAT(j,:)) denotes the maximum of the absolute values of the jth row of the matrix MAT. In this case, a diagonal element of U is small, indicating that MAT is singular or nearly singular.

On exit, if MAT is singular, the function returns a n-vector filled with nan() value.

A blocked algorithm is used to compute the factorization and solve the triangular systems. Furthermore, the computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2011:** Gaussian elimination. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 3, Issue 3, pp 230-238.

6.10.12 `function solve_lin ( mat, b, tol )`

Purpose

SOLVE_LIN solves a system of linear equations with several right hand sides

MAT * X = B

with a n-by-n coefficient matrix MAT. B is a n-by-nb matrix.

The function returns the n-by-nb solution matrix X, if MAT is not singular.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) MAT contains the coefficient matrix of the equation

MAT * X = B

MAT is not modified by the subroutine.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

B (INPUT) real(stnd), dimension(:,:) On entry, the right hand side matrix B. B is not modified by the subroutine.

The shape of B must verify: size(B, 1) = n .

TOL (INPUT, OPTIONAL) real(stnd) On entry, a relative tolerance used to indicate whether or not MAT is nearly singular. Tol should normally be choose as approximately the largest relative error in the elements of MAT. For example, if the elements of MAT are correct to about 4 significant figures, then TOL should be set to about 5 * 10**(-4). If TOL is supplied as less than EPS, where EPS is the relative machine precision, then the value EPS is used in place of TOL.

Default: EPS, the relative machine precision.

Further Details

The LU decomposition with partial pivoting and implicit row scaling of the matrix MAT is used to solve the linear system.

MAT is declared singular if a diagonal element of U is such that

abs(U(j,j)) <= n * norm(MAT(j,:)) * TOL

where norm(MAT(j,:)) denotes the maximum of the absolute values of the jth row of the matrix MAT. In this case, a diagonal element of U is small, indicating that MAT is singular or nearly singular.

On exit, if MAT is algorithmically singular, the function returns a n-by-nb matrix filled with nan() value.

A blocked algorithm is used to compute the factorization and solve the triangular systems. Furthermore, the computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2011:** Gaussian elimination. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 3, Issue 3, pp 230-238.

6.10.13 `subroutine lin_lu_solve ( mat, b, failure, tol, small )`

Purpose

LIN_LU_SOLVE solves a system of linear equations

MAT * X = B

with a n-by-n coefficient matrix MAT. B is a n-vector.

The LU decomposition with partial pivoting and implicit row scaling of the matrix MAT

P * MAT = L * U

where P is a permutation matrix, L is a n-by-n unit lower triangular matrix and U is a n-by-n upper triangular matrix, is used to solve the linear system.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, MAT contains the coefficient matrix of the equation

MAT * X = B

On exit, MAT is destroyed.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

B (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the right hand side vector B.

On exit, the solution vector X.

The shape of B must verify: size(B) = n .

FAILURE (OUTPUT) logical(lgl) On exit, if:

- FAILURE = true : MAT is algorithmically singular

- FAILURE = false: MAT is not singular.

TOL (INPUT, OPTIONAL) real(stnd) On entry, a relative tolerance used to indicate whether or not MAT is nearly singular. Tol should normally be choose as approximately the largest relative error in the elements of MAT. For example, if the elements of MAT are correct to about 4 significant figures, then TOL should be set to about 5 * 10**(-4). If TOL is supplied as less than EPS, where EPS is the relative machine precision, then the value EPS is used in place of TOL.

Default: EPS, the relative machine precision.

SMALL (INPUT, OPTIONAL) real(stnd) On entry, if the system is singular, replaces a diagonal term of the matrix U if it is smaller in magnitude than the value SMALL using the same sign as the diagonal term. An approximate solution based on this replacement is obtained if no overflow results. If SMALL is supplied as less than SAFMIN, the smallest number that can be reciprocated safely, then the value SAFMIN is used in place of SMALL.

Default: SAFMIN, the smallest number that can be reciprocated safely.

Further Details

MAT is declared singular if a diagonal element of U is such that

abs(U(j,j)) <= n * norm(MAT(j,:)) * TOL

where norm(MAT(j,:)) denotes the maximum of the absolute values of the jth row of the matrix MAT. In this case, a diagonal element of U is small, indicating that MAT is singular or nearly singular and FAILURE is set to true. Otherwise, FAILURE is set to false.

If MAT is algorithmically singular (FAILURE=true), the diagonal terms of U smaller in magnitude than the value SMALL have been replaced by SMALL, using the same sign as the diagonal terms, and the decomposition has been completed. An approximate solution based on this replacement is obtained if no overflow results.

A blocked algorithm is used to compute the factorization and solve the triangular systems. Furthermore, the computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2011:** Gaussian elimination. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 3, Issue 3, pp 230-238.

6.10.14 `subroutine lin_lu_solve ( mat, b, failure, tol, small )`

Purpose

LIN_LU_SOLVE solves a system of linear equations with several right hand sides

MAT * X = B

with a n-by-n coefficient matrix MAT. B is a n-by-nb matrix.

The LU decomposition with partial pivoting and implicit row scaling of the matrix MAT

P * MAT = L * U

where P is a permutation matrix, L is a n-by-n unit lower triangular matrix and U is a n-by-n upper triangular matrix, is used to solve the linear systems.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, MAT contains the coefficient matrix of the equation

MAT * X = B

On exit, MAT is destroyed.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

B (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the right hand side matrix B.

On exit, the solution matrix X.

The shape of B must verify: size(B, 1) = n .

FAILURE (OUTPUT) logical(lgl) On exit, if:

- FAILURE = true : MAT is algorithmically singular
- FAILURE = false: MAT is not singular.

TOL (INPUT, OPTIONAL) real(stnd) On entry, a relative tolerance used to indicate whether or not MAT is nearly singular. Tol should normally be choose as approximately the largest relative error in the elements of MAT. For example, if the elements of MAT are correct to about 4 significant figures, then TOL should be set to about 5 * 10**(-4). If TOL is supplied as less than EPS, where EPS is the relative machine precision, then the value EPS is used in place of TOL.

Default: EPS, the relative machine precision.

SMALL (INPUT, OPTIONAL) real(stnd) On entry, if the system is singular, replaces a diagonal term of the matrix U if it is smaller in magnitude than the value SMALL using the same sign as the diagonal term. Approximate solutions based on this replacement are obtained if no overflow results. If SMALL is supplied as less than SAFMIN, the smallest number that can be reciprocated safely, then the value SAFMIN is used in place of SMALL.

Default: SAFMIN, the smallest number that can be reciprocated safely.

Further Details

MAT is declared singular if a diagonal element of U is such that

abs(U(j,j)) <= n * norm(MAT(j,:)) * TOL

where norm(MAT(j,:)) denotes the maximum of the absolute values of the jth row of the matrix MAT. In this case, a diagonal element of U is small, indicating that MAT is singular or nearly singular and FAILURE is set to true. Otherwise, FAILURE is set to false.

If MAT is algorithmically singular (FAILURE=true), the diagonal terms of U smaller in magnitude than the value SMALL have been replaced by SMALL, using the same sign as the diagonal terms, and the decomposition has been completed. Approximate solutions based on this replacement are obtained if no overflow results.

A blocked algorithm is used to compute the factorization and solve the triangular systems. Furthermore, the computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2011:** Gaussian elimination. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 3, Issue 3, pp 230-238.

6.10.15 `subroutine lin_lu_solve ( mat, b, failure, x, tol, small )`

Purpose

LIN_LU_SOLVE solves a system of linear equations

MAT * X = B

with a n-by-n coefficient matrix MAT. B and X are n-vectors.

The LU decomposition with partial pivoting and implicit row scaling of the matrix MAT

P * MAT = L * U

where P is a permutation matrix, L is a n-by-n unit lower triangular matrix and U is a n-by-n upper triangular matrix, is used to solve the linear system.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) MAT contains the coefficient matrix of the equation

MAT * X = B

MAT is not modified by the subroutine.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

B (INPUT) real(stnd), dimension(:) On entry, the right hand side vector B. B is not modified by the subroutine.

The shape of B must verify: size(B) = n .

FAILURE (OUTPUT) logical(lgl) On exit, if:

- FAILURE = true : MAT is algorithmically singular
- FAILURE = false: MAT is not singular.

X (OUTPUT) real(stnd), dimension(:) On exit, the solution vector X.

The shape of X must verify: size(X) = n .

TOL (INPUT, OPTIONAL) real(stnd) On entry, a relative tolerance used to indicate whether or not MAT is nearly singular. Tol should normally be choose as approximately the largest relative error in the elements of MAT. For example, if the elements of MAT are correct to about 4 significant figures, then TOL should be set to about 5 * 10**(-4). If TOL is supplied as less than EPS, where EPS is the relative machine precision, then the value EPS is used in place of TOL.

Default: EPS, the relative machine precision.

SMALL (INPUT, OPTIONAL) real(stnd) On entry, if the system is singular, replaces a diagonal term of the matrix U if it is smaller in magnitude than the value SMALL using the same sign as the diagonal term. An approximate solution based on this replacement is obtained if no overflow results. If SMALL is supplied as less than SAFMIN, the smallest number that can be reciprocated safely, then the value SAFMIN is used in place of SMALL.

Default: SAFMIN, the smallest number that can be reciprocated safely.

Further Details

MAT is declared singular if a diagonal element of U is such that

abs(U(j,j)) <= n * norm(MAT(j,:)) * TOL

where norm(MAT(j,:)) denotes the maximum of the absolute values of the jth row of the matrix MAT. In this case, a diagonal element of U is small, indicating that MAT is singular or nearly singular and FAILURE is set to true. Otherwise, FAILURE is set to false.

If MAT is algorithmically singular (FAILURE=true), the diagonal terms of U smaller in magnitude than the value SMALL have been replaced by SMALL, using the same sign as the diagonal terms, and the decomposition has been completed. An approximate solution based on this replacement is obtained if no overflow results.

A blocked algorithm is used to compute the factorization and solve the triangular systems. Furthermore, the computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2011:** Gaussian elimination. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 3, Issue 3, pp 230-238.

6.10.16 `subroutine lin_lu_solve ( mat, b, failure, x, tol, small )`

Purpose

LIN_LU_SOLVE solves a system of linear equations with several right hand sides

MAT * X = B

with a n-by-n coefficient matrix MAT. B and X are n-by-nb matrices.

The LU decomposition with partial pivoting and implicit row scaling of the matrix MAT

P * MAT = L * U

where P is a permutation matrix, L is a n-by-n unit lower triangular matrix and U is a n-by-n upper triangular matrix, is used to solve the linear systems.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) MAT contains the coefficient matrix of the equation

MAT * X = B

MAT is not modified by the subroutine.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

B (INPUT) real(stnd), dimension(:,:) On entry, the right hand side matrix B. B is not modified by the subroutine.

The shape of B must verify: size(B, 1) = n .

FAILURE (OUTPUT) logical(lgl) On exit, if:

- FAILURE = true : MAT is algorithmically singular

- FAILURE = false: MAT is not singular.

X (OUTPUT) real(stnd), dimension(:,:) On exit, the solution matrix X.

The shape of X must verify:

- size(X, 1) = n ;
- size(X, 2) = size(B, 2) = nb .

TOL (INPUT, OPTIONAL) real(stnd) On entry, a relative tolerance used to indicate whether or not MAT is nearly singular. Tol should normally be choose as approximately the largest relative error in the elements of MAT. For example, if the elements of MAT are correct to about 4 significant figures, then TOL should be set to about 5 * 10**(-4). If TOL is supplied as less than EPS, where EPS is the relative machine precision, then the value EPS is used in place of TOL.

Default: EPS, the relative machine precision.

SMALL (INPUT, OPTIONAL) real(stnd) On entry, if the system is singular, replaces a diagonal term of the matrix U if it is smaller in magnitude than the value SMALL using the same sign as the diagonal term. Approximate solutions based on this replacement are obtained if no overflow results. If SMALL is supplied as less than SAFMIN, the smallest number that can be reciprocated safely, then the value SAFMIN is used in place of SMALL.

Default: SAFMIN, the smallest number that can be reciprocated safely.

Further Details

MAT is declared singular if a diagonal element of U is such that

abs(U(j,j)) <= n * norm(MAT(j,:)) * TOL

where norm(MAT(j,:)) denotes the maximum of the absolute values of the jth row of the matrix MAT. In this case, a diagonal element of U is small, indicating that MAT is singular or nearly singular and FAILURE is set to true. Otherwise, FAILURE is set to false.

If MAT is algorithmically singular (FAILURE=true), the diagonal terms of U smaller in magnitude than the value SMALL have been replaced by SMALL, using the same sign as the diagonal terms, and the decomposition has been completed. Approximate solutions based on this replacement are obtained if no overflow results.

A blocked algorithm is used to compute the factorization and solve the triangular systems. Furthermore, the computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2011:** Gaussian elimination. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 3, Issue 3, pp 230-238.

6.10.17 `subroutine chol_solve ( mat, invdiag, b, upper )`

Purpose

CHOL_SOLVE solves a system of linear equations

MAT * X = B

where MAT is a n-by-n symmetric positive definite matrix and B is a n-vector, using the CHOLESKY factorization MAT = U' * U or MAT = L * L', as computed by CHOL_CMP or GCHOL_CMP.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the triangular factor U or L from the Cholesky factorisation, as computed by CHOL_CMP, except for the main diagonal elements which are stored in reciprocal form in INVDIAG.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

INVDIAG (INPUT) real(stnd), dimension(:) On entry, INVDIAG contains the reciprocals of the actual diagonal elements of L or U, as computed by CHOL_CMP.

The shape of INVDIAG must verify: size(INVDIAG) = n .

B (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the right hand side vector B.

On exit, the solution vector X.

The shape of B must verify: size(B) = n .

UPPER (INPUT, OPTIONAL) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular is stored
- UPPER = false: Lower triangular is stored.

This argument must have the same value as used in CHOL_CMP or GCHOL_CMP subroutines for computing the Cholesky factorisation.

The default is true.

6.10.18 `subroutine chol_solve ( mat, invdiag, b, upper )`

Purpose

CHOL_SOLVE solves a system of linear equations with several right hand sides

MAT * X = B

where MAT is a n-by-n symmetric positive (semi)-definite matrix and B is a n-by-nb matrix, using the CHOLESKY factorization MAT = U' * U or MAT = L * L' as computed by CHOL_CMP or GCHOL_CMP.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the triangular factor U or L from the Cholesky factorisation, as computed by CHOL_CMP, except for the main diagonal elements which are stored in reciprocal form in INVDIAG.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

INVDIAG (INPUT) real(stnd), dimension(:) On entry, INVDIAG contains the reciprocals of the actual diagonal elements of L or U, as computed by CHOL_CMP.

The shape of INVDIAG must verify: size(INVDIAG) = n .

B (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the right hand side matrix B.

On exit, the solution matrix X.

The shape of B must verify: size(B, 1) = n .

UPPER (INPUT, OPTIONAL) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular is stored
- UPPER = false: Lower triangular is stored.

This argument must have the same value as used in CHOL_CMP or GCHOL_CMP subroutines for computing the Cholesky factorisation.

The default is true.

Further Details

The computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2009:** Cholesky factorization. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 1, pp 251-254.

6.10.19 `subroutine triang_solve ( mat, b, upper, trans )`

Purpose

TRIANG_SOLVE solves a triangular system of the form

MAT * X = B or MAT' * X = B,

where MAT is a triangular matrix of order n, and B is an n-vector.

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) The triangular matrix MAT. If:

- UPPER = true or is absent: The leading n-by-n upper triangular part of the array MAT contains the upper triangular matrix, and the strictly lower triangular part of MAT is not referenced.
- UPPER = false: The leading n-by-n lower triangular part of the array MAT contains the lower triangular matrix, and the strictly upper triangular part of MAT is not referenced.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

B (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the right hand side vector B.

On exit, the solution vector X.

The shape of B must verify: size(B) = n .

UPPER (INPUT, OPTIONAL) logical(lgl) Specifies whether MAT is upper or lower triangular. If:

- UPPER = true : MAT is upper triangular
- UPPER = false: MAT is lower triangular.

The default is true.

TRANS (INPUT, OPTIONAL) logical(lgl) Specifies the form of the system of equations. If:

- TRANS = true : MAT' * X = B (Transpose)
- TRANS = false: MAT * X = B (No transpose)

The default is false.

6.10.20 `subroutine triang_solve ( mat, b, scal, upper, trans )`

Purpose

TRIANG_SOLVE solves a nonsingular triangular linear system of the form

MAT * X = B or MAT' * X = B,

where MAT is a triangular matrix of order n, and B is an n-vector.

The matrix MAT is assumed to be ill-conditioned, and frequent rescalings are carried out in order to avoid overflow. However, no test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) The nonsingular triangular matrix MAT. If:

- UPPER = true or is absent: The leading n-by-n upper triangular part of the array MAT contains the upper triangular matrix, and the strictly lower triangular part of MAT is not referenced.
- UPPER = false: The leading n-by-n lower triangular part of the array MAT contains the lower triangular matrix, and the strictly upper triangular part of MAT is not referenced.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

B (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the right hand side vector B.

On exit, the scaled solution vector X.

The shape of B must verify: size(B) = n .

SCAL (OUTPUT) real(stnd) On exit, SCAL is a scaling factor introduced in order to avoid overflow. The solution of the given system of equations is B(:)/SCAL . Note that SCAL may be negative.

UPPER (INPUT, OPTIONAL) logical(lgl) Specifies whether MAT is upper or lower triangular. If:

- UPPER = true : MAT is upper triangular
- UPPER = false: MAT is lower triangular.

The default is true.

TRANS (INPUT, OPTIONAL) logical(lgl) Specifies the form of the system of equations. If:

- TRANS = true : MAT' * X = B (Transpose)
- TRANS = false: MAT * X = B (No transpose)

The default is false.

6.10.21 subroutine triang_solve (mat, b, upper, trans)

Purpose

TRIANG_SOLVE solves a triangular system of the form

MAT * X = B or MAT' * X = B,

where MAT is a triangular matrix of order n, and B is an n-by-nb matrix.

No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) The triangular matrix MAT. If:

- UPPER = true or is absent: The leading n-by-n upper triangular part of the array MAT contains the upper triangular matrix, and the strictly lower triangular part of MAT is not referenced.
- UPPER = false: The leading n-by-n lower triangular part of the array MAT contains the lower triangular matrix, and the strictly upper triangular part of MAT is not referenced.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

B (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the right hand side matrix B.

On exit, the solution matrix X.

The shape of B must verify: size(B, 1) = n .

UPPER (INPUT, OPTIONAL) logical(lgl) Specifies whether MAT is upper or lower triangular. If:

- UPPER = true : MAT is upper triangular
- UPPER = false: MAT is lower triangular.

The default is true.

TRANS (INPUT, OPTIONAL) logical(lgl) Specifies the form of the system of equations. If:

- TRANS = true : MAT' * X = B (Transpose)
- TRANS = false: MAT * X = B (No transpose)

The default is false.

Further Details

The computations are parallelized if OPENMP is used.

6.10.22 subroutine triang_solve (mat, b, scal, upper, trans)

Purpose

TRIANG_SOLVE solves a nonsingular triangular linear system of the form

MAT * X = B or MAT' * X = B,

where MAT is a triangular matrix of order n, and B is an n-by-nb matrix.

The matrix MAT is assumed to be ill-conditioned, and frequent rescalings are carried out in order to avoid overflow. However, no test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) The nonsingular triangular matrix MAT. If:

- UPPER = true or is absent: The leading n-by-n upper triangular part of the array MAT contains the upper triangular matrix, and the strictly lower triangular part of MAT is not referenced.
- UPPER = false: The leading n-by-n lower triangular part of the array MAT contains the lower triangular matrix, and the strictly upper triangular part of MAT is not referenced.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

B (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the right hand side matrix B.

On exit, the scaled solution matrix X.

The shape of B must verify:

- size(B, 1) = n ,
- size(B, 2) = size(SCAL) = nb .

SCAL (OUTPUT) real(stnd), dimension(:) On exit, SCAL is a scaling vector introduced in order to avoid overflow. The solution of the given system of equations is B/spread(SCAL,dim=1, ncopies=n) . Note that elements of SCAL may be negative.

The size of SCAL must verify: size(SCAL) = size(B, 2) = nb .

UPPER (INPUT, OPTIONAL) logical(lgl) Specifies whether MAT is upper or lower triangular. If:

- UPPER = true : MAT is upper triangular
- UPPER = false: MAT is lower triangular.

The default is true.

TRANS (INPUT, OPTIONAL) logical(lgl) Specifies the form of the system of equations. If:

- TRANS = true : MAT' * X = B (Transpose)
- TRANS = false: MAT * X = B (No transpose)

The default is false.

Further Details

The computations are parallelized if OPENMP is used.

6.10.23 `subroutine comp_inv ( mat, failure, tol )`

Purpose

COMP_INV computes, in place, the inverse of a matrix MAT.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, MAT contains the matrix to be inverted.

On exit, MAT is replaced by its inverse.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

FAILURE (OUTPUT) logical(lgl) On exit, if:

- FAILURE = true : MAT is algorithmically singular.
- FAILURE = false: MAT has been inverted.

TOL (INPUT, OPTIONAL) real(stnd) On entry, a relative tolerance used to indicate whether or not MAT is nearly singular. Tol should normally be choose as approximately the largest relative error in the elements of MAT. For example, if the elements of MAT are correct to about 4 significant figures, then TOL should be set to about 5 * 10**(-4). If TOL is supplied as less than EPS, where EPS is the relative machine precision, then the value EPS is used in place of TOL.

Default: EPS, the relative machine precision.

Further Details

The LU decomposition with partial pivoting and implicit row scaling of the matrix MAT is used to compute the inverse.

MAT is declared singular if a diagonal element of U is such that

abs(U(j,j)) <= n * norm(MAT(j,:)) * TOL

where norm(MAT(j,:)) denotes the maximum of the absolute values of the jth row of the matrix MAT. In this case, a diagonal element of U is small, indicating that MAT is singular or nearly singular and FAILURE is set to true. Otherwise, FAILURE is set to false.

On exit, if FAILURE=true, MAT is filled with nan() value.

The computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2011:** Gaussian elimination. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 3, Issue 3, pp 230-238.

6.10.24 `subroutine comp_inv ( mat, failure, matinv, tol )`

Purpose

COMP_INV computes the inverse of a matrix MAT.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, MAT contains the matrix to be inverted.

On exit, MAT is replaced by the LU decomposition of a rowwise permutation of MAT.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

FAILURE (OUTPUT) logical(lgl) On exit, if:

- FAILURE = true : MAT is algorithmically singular.
- FAILURE = false: MAT has been inverted.

MATINV (OUTPUT) real(stnd), dimension(:,:)

On exit, if MAT is not singular, MATINV contains the inverse of MAT.

The shape of MATINV must verify: size(MATINV, 1) = size(MATINV, 2) = n .

TOL (INPUT, OPTIONAL) real(stnd) On entry, a relative tolerance used to indicate whether or not MAT is nearly singular. Tol should normally be choose as approximately the largest relative error in the elements of MAT. For example, if the elements of MAT are correct to about 4 significant figures, then TOL should be set to about 5 * 10**(-4). If TOL is supplied as less than EPS, where EPS is the relative machine precision, then the value EPS is used in place of TOL.

Default: EPS, the relative machine precision.

Further Details

MAT is modified by COMP_INV.

The LU decomposition with partial pivoting and implicit row scaling of the matrix MAT is used to compute the inverse.

MAT is declared singular if a diagonal element of U is such that

abs(U(j,j)) <= n * norm(MAT(j,:)) * TOL

where norm(MAT(j,:)) denotes the maximum of the absolute values of the jth row of the matrix MAT. In this case, a diagonal element of U is small, indicating that MAT is singular or nearly singular and FAILURE is set to true. Otherwise, FAILURE is set to false.

On exit, if FAILURE=true, MATINV is filled with nan() value.

The computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2011:** Gaussian elimination. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 3, Issue 3, pp 230-238.

6.10.25 `function inv ( mat, tol )`

Purpose

INV computes the inverse of a real matrix MAT,

MAT * INV(MAT) = I

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, MAT contains the matrix to be inverted. MAT is not modified by the function.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

TOL (INPUT, OPTIONAL) real(stnd) On entry, a relative tolerance used to indicate whether or not MAT is nearly singular. Tol should normally be choose as approximately the largest relative error in the elements of MAT. For example, if the elements of MAT are correct to about 4 significant figures, then TOL should be set to about 5 * 10**(-4). If TOL is supplied as less than EPS, where EPS is the relative machine precision, then the value EPS is used in place of TOL.

Default: EPS, the relative machine precision.

Further Details

MAT is not modified by function INV.

The LU decomposition with partial pivoting and implicit row scaling of the matrix MAT is used to compute the inverse.

MAT is declared singular if a diagonal element of U is such that

abs(U(j,j)) <= n * norm(MAT(j,:)) * TOL

where norm(MAT(j,:)) denotes the maximum of the absolute values of the jth row of the matrix MAT. In this case, a diagonal element of U is small, indicating that MAT is singular or nearly singular.

On exit, if MAT is algorithmically singular, the function INV returns a n-by-n matrix filled with nan() value.

The computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2011:** Gaussian elimination. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 3, Issue 3, pp 230-238.

6.10.26 subroutine comp_sym_inv (mat, failure, upper, fill, tol)

Purpose

COMP_SYM_INV computes, in place, the inverse of a real symmetric positive definite matrix MAT using the Cholesky factorization MAT = U' * U or MAT = L * L'.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix to be inverted:

- If UPPER = true or is absent: The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix and the strictly lower triangular part of MAT is not referenced.
- If UPPER = false: The leading n-by-n lower triangular part of MAT contains the lower triangular part of the matrix and the strictly upper triangular part of MAT is not referenced.

On exit:

- If FILL = true or is absent: The (symmetric) inverse of MAT overwrites MAT.
- If FILL = false: The upper (if UPPER= true) or lower (if UPPER= false) triangle of the (symmetric) inverse of MAT, overwrites the input upper or lower triangular part of MAT and the other part of MAT is not modified.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

FAILURE (OUTPUT) logical(lgl) On exit, if:

- FAILURE = true : MAT is algorithmically not positive definite.
- FAILURE = false: MAT has been inverted.

UPPER (INPUT, OPTIONAL) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular is stored
- UPPER = false: Lower triangular is stored.

The default is true.

FILL (INPUT, OPTIONAL) logical(lgl) On entry, when argument FILL is present, FILL is used as follows. If:

- FILL= true and UPPER= true, the lower triangle of MAT is filled on exit.
- FILL= true and UPPER= false, the upper triangle of MAT is filled on exit.
- FILL= false, the lower (UPPER= true) or upper (UPPER= false) triangle of MAT is not filled and not modified on exit.

The default is true.

TOL (INPUT, OPTIONAL) real(stnd) Tolerance used to test the matrix for positive-definiteness. TOL is used as a multiplying factor for determining effective zero for pivots. TOL must be greater or equal to zero, otherwise the default value is used.

The default value is the machine precision multiplied by n.

Further Details

MAT is declared not positive definite if during the j-th stage of the factorization of MAT, a pivot, PIV(j), is such that

PIV(j) <= MAT(j,j) * TOL

In this case, the leading minor of order j of MAT is declared not positive definite, the Cholesky factorization is not completed and, on exit of COMP_SYM_INV, FAILURE is set to true.

On exit, if FAILURE=true:

- The upper or lower triangle of MAT is filled with nan() value if FILL=false.
- The matrix MAT is filled with nan() value if FILL=true.

A blocked algorithm is used to compute the Cholesky factorization. Furthermore, the computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.10.27 `subroutine comp_sym_inv ( mat, failure, matinv, upper, fill, tol )`

Purpose

COMP_SYM_INV computes the inverse of a real symmetric positive definite matrix MAT using the Cholesky factorization MAT = U' * U or MAT = L * L'.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix to be inverted.

- If UPPER = true or is absent: The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix and the strictly lower triangular part of MAT is not referenced.
- If UPPER = false: The leading n-by-n lower triangular part of MAT contains the lower triangular part of the matrix and the strictly upper triangular part of MAT is not referenced.

MAT is not modified by the subroutine.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

FAILURE (OUTPUT) logical(lgl) On exit, if:

- FAILURE = true : MAT is algorithmically not positive definite.
- FAILURE = false: MAT has been inverted.

MATINV (OUTPUT) real(stnd), dimension(:,:) On exit:

- If FILL = true or is absent: The (symmetric) inverse of MAT.
- If FILL = false: The upper (if UPPER=true) or lower (if UPPER=false) triangle of the (symmetric) inverse of MAT, is stored in the upper or lower triangular part of the matrix MATINV and the other part of MATINV is not modified.

The shape of MATINV must verify: size(MATINV, 1) = size(MATINV, 2) = n .

UPPER (INPUT, OPTIONAL) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular is stored
- UPPER = false: Lower triangular is stored.

The default is true.

FILL (INPUT, OPTIONAL) logical(lgl) On entry, when argument FILL is present, FILL is used as follows. If:

- FILL= true and UPPER= true, the lower triangle of MATINV is filled on exit.
- FILL= true and UPPER= false, the upper triangle of MATINV is filled on exit.
- FILL= false, the lower (UPPER= true) or upper (UPPER= false) triangle of MATINV is not filled and not modified on exit.

The default is true.

TOL (INPUT, OPTIONAL) real(stnd) Tolerance used to test the matrix for positive-definiteness. TOL is used as a multiplying factor for determining effective zero for pivots. TOL must be greater or equal to zero, otherwise the default value is used.

The default value is the machine precision multiplied by n.

Further Details

MAT is not modified by COMP_SYM_INV.

MAT is declared not positive definite if during the j-th stage of the factorization of MAT, a pivot, PIV(j), is such that

PIV(j) <= MAT(j,j) * TOL

In this case, the leading minor of order j of MAT is declared not positive definite, the Cholesky factorization is not completed and, on exit of COMP_SYM_INV, FAILURE is set to true.

On exit, if FAILURE=true:

- The upper or lower triangle of MATINV is filled with nan() value if FILL=false.
- The matrix MATINV is filled with nan() value if FILL=true.

A blocked algorithm is used to compute the factorization. Furthermore, the computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.10.28 `function sym_inv ( mat, upper, tol )`

Purpose

SYM_INV computes the inverse of a real symmetric positive definite matrix MAT using the Cholesky factorization MAT = U' * U or MAT = L * L'.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix to be inverted. If:

- If UPPER = true or is absent: The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix and the strictly lower triangular part of MAT is not referenced.
- If UPPER = false: The leading n-by-n lower triangular part of MAT contains the lower triangular part of the matrix and the strictly upper triangular part of MAT is not referenced.

MAT is not modified by the function.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

UPPER (INPUT, OPTIONAL) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular
- UPPER = false: Lower triangular.

The default is true.

TOL (INPUT, OPTIONAL) real(stnd) Tolerance used to test the matrix for positive-definiteness. TOL is used as a multiplying factor for determining effective zero for pivots. TOL must be greater or equal to zero, otherwise the default value is used.

The default value is the machine precision multiplied by n.

Further Details

MAT is not modified by function SYM_INV.

The (symmetric) inverse of MAT is returned if MAT is positive definite.

MAT is declared not positive definite if during the j-th stage of the Cholesky factorization of MAT, a pivot, PIV(j), is such that

PIV(j) <= MAT(j,j) * TOL

In this case, the leading minor of order j of MAT is declared not positive definite.

On exit, if MAT is algorithmically not positive definite, SYM_INV returns a matrix filled with nan() value.

A blocked algorithm is used to compute the factorization. Furthermore, the computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.10.29 `subroutine comp_sym_ginv ( mat, failure, krank, upper, fill, tol )`

Purpose

COMP_SYM_GINV computes, in place, the (generalized) inverse of a real symmetric positive semidefinite matrix MAT using the Cholesky factorization MAT = U' * U or MAT = L * L'.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix to be inverted. If:

- UPPER = true or is absent: The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix and the strictly lower triangular part of MAT is not referenced.
- UPPER = false: The leading n-by-n lower triangular part of MAT contains the lower triangular part of the matrix and the strictly upper triangular part of MAT is not referenced.

On exit, if:

- FILL = true or is absent: The (symmetric) inverse of MAT overwrites MAT.
- FILL = false: The upper or lower triangle of the (symmetric) inverse of MAT, overwrites the input upper or lower triangular part of the matrix MAT and the other part of MAT is not modified.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

FAILURE (OUTPUT) logical(lgl) On exit, if:

- FAILURE = true : MAT is algorithmically not positive semidefinite.
- FAILURE = false: MAT has been inverted.

KRANK (OUTPUT) integer(i4b) On exit, KRANK contains the effective rank of MAT, which is defined as the number of nonzero elements of the diagonal of L or U. Note that KRANK may be different from the true rank of MAT. See the reference (2) for details.

UPPER (INPUT, OPTIONAL) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular is stored
- UPPER = false: Lower triangular is stored.

The default is true.

FILL (INPUT, OPTIONAL) logical(lgl) On entry, when argument FILL is present, FILL is used as follows. If:

- FILL= true and UPPER= true, the lower triangle of MAT is filled on exit.
- FILL= true and UPPER= false, the upper triangle of MAT is filled on exit.
- FILL= false, the lower (UPPER= true) or upper (UPPER= false) triangle of MAT is not filled on exit.

The default is true.

TOL (INPUT, OPTIONAL) real(stnd) Tolerance used to test MAT for positive-semidefiniteness. TOL is used as a multiplying factor for determining effective zero for pivots. TOL must be greater or equal to zero, otherwise the default value is used.

The default value is the machine precision multiplied by n.

Further Details

If MAT is positive semidefinite, the subroutine computes a generalized inverse of MAT. GMAT is a generalized inverse of MAT if

MAT * GMAT * MAT = MAT and GMAT * MAT * GMAT = GMAT

See description of subroutine GCHOL_CMP2 for more details. The subroutine also computes and returns an estimate of the effective rank of MAT in the argument RANK. Note that KRANK may be different from the true rank of MAT. See the reference (2) for details.

MAT is declared not positive semidefinite if during the j-th stage of the Cholesky factorization of MAT, a pivot, PIV(j), is such that

PIV(j) <= - abs(MAT(j,j) * TOL)

In this case, the leading minor of order j of MAT is declared not positive semidefinite, the Cholesky factorization is not completed and on exit of COMP_SYM_GINV, FAILURE is set to true.

On exit, if FAILURE=true:

- KRANK is set to -1,
- The upper or lower triangle of MAT is filled with nan() value if FILL=false.
- The matrix MAT is filled with nan() value if FILL=true.

A blocked algorithm is used to compute the Cholesky factorization. Furthermore, the computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2009:** Cholesky factorization. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 1, pp 251-254.

6.10.30 subroutine comp_sym_ginv (mat, failure, krank, matinv, upper, fill, tol)

Purpose

COMP_SYM_GINV computes the (generalized) inverse of a real symmetric positive semidefinite matrix MAT using the Cholesky factorization MAT = U' * U or MAT = L * L'.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the symmetric matrix to be inverted. If:

- UPPER = true or is absent: The leading n-by-n upper triangular part of MAT contains the upper triangular part of the matrix and the strictly lower triangular part of MAT is not referenced.
- If UPPER = false: The leading n-by-n lower triangular part of MAT contains the lower triangular part of the matrix and the strictly upper triangular part of MAT is not referenced.

MAT is not modified by the subroutine.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

FAILURE (OUTPUT) logical(lgl) On exit, if:

-FAILURE = true : MAT is algorithmically not positive semidefinite. -FAILURE = false: MAT has been inverted.

KRANK (OUTPUT) integer(i4b) On exit, KRANK contains the effective rank of MAT, which is defined as the number of nonzero elements of the diagonal of L or U. Note that KRANK may be different from the true rank of MAT. See the reference (2) for details.

MATINV (OUTPUT) real(stnd), dimension(:,:) On exit, if:

- FILL = true or is absent: The (symmetric) (generalized) inverse of MAT.
- If FILL = false: The upper (if UPPER=true) or lower (if UPPER=false) triangle of the (symmetric) (generalized) inverse of MAT, is stored in the upper or lower triangular part of the matrix MATINV and the other part of MATINV is not modified.

The shape of MATINV must verify: size(MATINV, 1) = size(MATINV, 2) = n .

UPPER (INPUT, OPTIONAL) logical(lgl) Specifies whether the upper or lower triangular part of the symmetric matrix MAT is stored. If:

- UPPER = true : Upper triangular is stored
- UPPER = false: Lower triangular is stored.

The default is true.

FILL (INPUT, OPTIONAL) logical(lgl) On entry, when argument FILL is present, FILL is used as follows. If:

- FILL= true and UPPER= true, the lower triangle of MATINV is filled on exit.
- FILL= true and UPPER= false, the upper triangle of MATINV is filled on exit.
- FILL= false, the lower (UPPER= true) or upper (UPPER= false) triangle of MATINV is not filled on exit.

The default is true.

TOL (INPUT, OPTIONAL) real(stnd) Tolerance used to test MAT for positive-semidefiniteness. TOL is used as a multiplying factor for determining effective zero for pivots. TOL must be greater or equal to zero, otherwise the default value is used.

The default value is the machine precision multiplied by n.

Further Details

MAT is not modified by COMP_SYM_GINV.

If MAT is positive semidefinite, the subroutine computes a generalized inverse of MAT. MATINV is a generalized inverse of MAT if

MAT * MATINV * MAT = MAT and MATINV * MAT * MATINV = MATINV

See description of subroutine GCHOL_CMP2 for more details. The subroutine also computes and returns an estimate of the rank of MAT in the argument RANK. Note that KRANK may be different from the true rank of MAT. See the reference (2) for details.

MAT is declared not positive semidefinite if during the j-th stage of the Cholesky factorization of MAT, a pivot, PIV(j), is such that

PIV(j) <= - abs(MAT(j,j) * TOL)

In this case, the leading minor of order j of MAT is declared not positive semidefinite and on exit of COMP_SYM_GINV, FAILURE is set to true.

On exit, if FAILURE=true:

- KRANK is set to -1,
- The upper or lower triangle of MATINV is filled with nan() value if FILL=false,
- The matrix MATINV is filled with nan() value if FILL=true.

A blocked algorithm is used to compute the Cholesky factorization. Furthermore, the computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2009:** Cholesky factorization. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 1, pp 251-254.

6.10.31 `subroutine comp_triang_inv ( mat, upper )`

Purpose

COMP_TRIANG_INV computes, in place, the inverse of a real upper or lower triangular matrix MAT.

On entry, if MAT is algorithmically singular, diagonal terms smaller in magnitude than the value SAFMIN (the smallest number that can be reciprocated safely) are replaced by SAFMIN using the same sign as the diagonal term. An approximate solution based on this replacement is then obtained.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the triangular matrix to be inverted.

- If UPPER = true or is absent: The leading n-by-n upper triangular part of MAT contains the upper triangular matrix and the strictly lower triangular part of MAT is not referenced.
- If UPPER = false: The leading n-by-n lower triangular part of MAT contains the lower triangular matrix and the strictly upper triangular part of MAT is not referenced.

On exit, the (triangular) inverse of the original matrix in the same storage format.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

UPPER (INPUT, OPTIONAL) logical(lgl) Specifies whether the matrix MAT is upper or lower triangular. If:

- UPPER = true : MAT is Upper triangular,
- UPPER = false: MAT is Lower triangular.

The default is true.

Further Details

The computations are not parallelized even if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.10.32 `subroutine comp_triang_inv ( mat, matinv, upper )`

Purpose

COMP_TRIANG_INV computes the inverse of a real upper or lower triangular matrix MAT.

On entry, if MAT is algorithmically singular, diagonal terms smaller in magnitude than the value SAFMIN (the smallest number that can be reciprocated safely) are replaced by SAFMIN using the same sign as the diagonal term. An approximate solution based on this replacement is then obtained.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the triangular matrix to be inverted. If:

- UPPER = true or is absent: The leading n-by-n upper triangular part of MAT contains the upper triangular matrix and the strictly lower triangular part of MAT is not referenced.
- UPPER = false: The leading n-by-n lower triangular part of MAT contains the lower triangular matrix and the strictly upper triangular part of MAT is not referenced.

MAT is not modified by the subroutine.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

MATINV (OUTPUT) real(stnd), dimension(:,:) On exit, the (triangular) inverse of the original matrix in the same storage format.

The shape of MATINV must verify: size(MATINV, 1) = size(MATINV, 2) = n .

UPPER (INPUT, OPTIONAL) logical(lgl) Specifies whether the matrix MAT is upper or lower triangular. If:

- UPPER = true : MAT is upper triangular,
- UPPER = false: MAT is lower triangular.

The default is true.

Further Details

The computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.10.33 `subroutine comp_uut_ltl ( mat, upper, fill )`

Purpose

COMP_UUT_LTL computes, in place, the product U * U' or L' * L, where the triangular factor U or L is stored in the upper or lower triangular part of MAT.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, if:

- UPPER = true or is absent: The leading n-by-n upper triangular part of MAT contains the upper triangular factor U and the strictly lower triangular part of MAT is not referenced.
- UPPER = false: The leading n-by-n lower triangular part of MAT contains the lower triangular factor L and the strictly upper triangular part of MAT is not referenced.

On exit, if:

- If FILL = true or is absent: The product U * U' or L' * L overwrites MAT.
- If FILL = false and UPPER = true or is absent: The leading n-by-n upper triangular part of MAT is overwritten with the upper triangular part of the product U * U' and the strictly lower triangular part of MAT is not referenced.
- If FILL = false and UPPER = false: The leading n-by-n lower triangular part of MAT is overwritten with the lower triangular part of the product L' * L and the strictly upper triangular part of MAT is not referenced.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

UPPER (INPUT, OPTIONAL) logical(lgl) Specifies whether the triangular factor stored in matrix MAT is upper or lower triangular. If:

- UPPER = true : Upper triangular is stored
- UPPER = false: Lower triangularis stored.

The default is true.

FILL (INPUT, OPTIONAL) logical(lgl) On entry, when argument FILL is present, FILL is used as follows. If:

- FILL= true and UPPER= true, the lower triangle of MAT is filled on exit.
- FILL= true and UPPER= false, the upper triangle of MAT is filled on exit.
- FILL= false, the lower (UPPER= true) or upper (UPPER= false) triangle of MAT is not filled on exit.

The default is true.

Further Details

The computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.10.34 `subroutine comp_uut_ltl ( mat, prod, upper, fill )`

Purpose

COMP_UUT_LTL computes the product U * U' or L' * L, where the triangular factor U or L is stored in the upper or lower triangular part of MAT.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the triangular factor U or L.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

PROD (OUTPUT) real(stnd), dimension(:,:) On exit, if:

- FILL = true or is absent: The product U * U' or L' * L .
- FILL = false and UPPER = true or is absent: The leading n-by-n upper triangular part of PROD contains the upper triangular part of the product U*Ut and the strictly lower triangular part of PROD is not referenced.
- FILL = false and UPPER = false: The leading n-by-n lower triangular part of PROD contains the lower triangular part of the product Lt*L and the strictly upper triangular part of PROD is not referenced.

The shape of PROD must verify: size(PROD, 1) = size(PROD, 2) = n .

UPPER (INPUT, OPTIONAL) logical(lgl) Specifies whether the triangular factor stored in matrix MAT is the upper or lower triangle. If:

- UPPER = true : Upper triangular is stored
- UPPER = false: Lower triangular is stored.

The default is true.

FILL (INPUT, OPTIONAL) logical(lgl) On entry, when argument FILL is present, FILL is used as follows. If:

- FILL= true and UPPER= true, the lower triangle of PROD is filled on exit.
- FILL= true and UPPER= false, the upper triangle of PROD is filled on exit.

- FILL= false, the lower (UPPER= true) or upper (UPPER= false) triangle of PROD is not filled on exit.

The default is true.

Further Details

The computations are parallelized if OPENMP is used.

6.10.35 subroutine comp_det (mat, det, tol, man_det, exp_det)

Purpose

COMP_DET computes the determinant of a real matrix MAT

DET = determinant(MAT)

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real matrix MAT.

On exit, MAT is destroyed.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

DET (OUTPUT) real(stnd) On exit, the determinant of MAT.

TOL (INPUT, OPTIONAL) real(stnd) On entry, a relative tolerance used to indicate whether or not MAT is nearly singular. Tol should normally be choose as approximately the largest relative error in the elements of MAT. For example, if the elements of MAT are correct to about 4 significant figures, then TOL should be set to about 5 * 10**(-4). If TOL is supplied as less than EPS, where EPS is the relative machine precision, then the value EPS is used in place of TOL.

Default: EPS, the relative machine precision.

MAN_DET (OUTPUT, OPTIONAL) real(stnd) On exit, the mantissa of the determinant of MAT.

EXP_DET (OUTPUT, OPTIONAL) integer(i4b) On exit, the exponent of the determinant of MAT.

Further Details

The LU decomposition with partial pivoting and implicit row scaling of the matrix MAT is used to compute the determinant.

MAT is declared singular if a diagonal element of U is such that

abs(U(j,j)) <= n * norm(MAT(j,:)) * TOL

where norm(MAT(j,:)) denotes the maximum of the absolute values of the jth row of the matrix MAT. In this case, a diagonal element of U is small, indicating that MAT is singular or nearly singular and DET is set to zero.

On exit:

- DET = nan() if MAT is a zero sized matrix.
- DET = scale(MAN_DET, EXP_DET) if minexponent(DET) <= EXP_DET <= maxexponent(DET)
- DET = sign(0, MAN_DET) if EXP_DET < minexponent(DET)

- DET = sign(huge(DET), MAN_DET) if maxexponent(DET) < EXP_DET

A blocked algorithm is used to compute the LU factorization. Furthermore, the computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2011:** Gaussian elimination. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 3, Issue 3, pp 230-238.

6.10.36 `function det ( mat, tol )`

Purpose

DET computes the determinant of a real matrix MAT

determinant(MAT) = DET(MAT)

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the real matrix MAT. MAT is not modified by the routine.

The shape of MAT must verify: size(MAT, 1) = size(MAT, 2) = n .

TOL (INPUT, OPTIONAL) real(stnd) On entry, a relative tolerance used to indicate whether or not MAT is nearly singular. Tol should normally be choose as approximately the largest relative error in the elements of MAT. For example, if the elements of MAT are correct to about 4 significant figures, then TOL should be set to about 5 * 10**(-4). If TOL is supplied as less than EPS, where EPS is the relative machine precision, then the value EPS is used in place of TOL.

Default: EPS, the relative machine precision.

Further Details

The LU decomposition with partial pivoting and implicit row scaling of the matrix MAT is used to compute the determinant.

MAT is declared singular if a diagonal element of U is such that

abs(U(j,j)) <= n * norm(MAT(j,:)) * TOL

where norm(MAT(j,:)) denotes the maximum of the absolute values of the jth row of the matrix MAT. In this case, a diagonal element of U is small, indicating that MAT is singular or nearly singular and DET returns the value zero.

If MAT is a zero sized matrix, DET(MAT) = nan().

A blocked algorithm is used to compute the LU factorization. Furthermore, the computations are parallelized if OPENMP is used.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Higham, N.J., 2011:** Gaussian elimination. Wiley Interdisciplinary Reviews: Computational Statistics, Vol. 3, Issue 3, pp 230-238.

6.10.37 `subroutine sym_trid_cmp ( d, e, sub, diag, sup1, sup2, perm, tol )`

Purpose

SYM_TRID_CMP factorizes an n by n symmetric tridiagonal matrix T as

T = P * L * U

where P is a permutation matrix, L is a unit lower tridiagonal matrix with at most one non-zero subdiagonal elements per column and U is an upper triangular matrix with at most two non-zero superdiagonal elements per column.

The factorization is obtained by Gaussian elimination with partial pivoting and implicit row scaling.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the symmetric tridiagonal matrix T.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the symmetric tridiagonal matrix T and E(n) is arbitrary .

The size of E must be size(E) = size(D) = n .

SUB (OUTPUT) real(stnd), dimension(:) On exit, SUB(:n-1) contains the n-1 subdiagonal elements of the lower triangular matrix L of the factorization of T, SUB(n) is arbitrary .

The size of SUB must verify: size(SUB) = size(D) = n .

DIAG (OUTPUT) real(stnd), dimension(:) On exit, DIAG(:) contains the n diagonal elements of the upper triangular matrix U of the factorization of T.

The size of DIAG must verify: size(DIAG) = size(D) = n .

SUP1 (OUTPUT) real(stnd), dimension(:) On exit, SUP1(:n-1) contains the n-1 superdiagonal elements of the upper triangular matrix U of the factorization of T, SUP1(n) is arbitrary .

The size of SUP1 must verify: size(SUP1) = size(D) = n .

SUP2 (OUTPUT) real(stnd), dimension(:) On exit, SUP2(:n-2) contains the n-2 second superdiagonal elements of the upper triangular matrix U of the factorization of T, SUP2(n-1:n) is arbitrary .

The size of SUP2 must verify: size(SUP2) = size(D) = n .

PERM (OUTPUT) logical(lgl), dimension(:) On exit, PERM(:n-1) contains details of the permutation matrix P(j):

- if, an interchange occurred at the kth step of the elimination in the factorization of T, then PERM(k) = TRUE
- otherwise, PERM(k) = FALSE.

The element PERM(n) is set to TRUE, if there is an integer l such that

abs(U(l,l)) <= norm(T(l)) * TOL,

where norm(T(l)) denotes the sum of the absolute values of the lth row of the matrix T. If no such l exists then PERM(n) is returned as FALSE.

If PERM(n) is returned as TRUE, then a diagonal element of U is small, indicating that T is singular or nearly singular.

The size of PERM must verify: size(PERM) = size(D) = n .

TOL (INPUT, OPTIONAL) real(stnd) On entry, a relative tolerance used to indicate whether or not the matrix T is nearly singular. TOL should normally be choose as approximately the largest relative error in the elements of T. For example, if the elements of T are correct to about 4 significant figures, then TOL should be set to about 5 * 10**(-4). If TOL is supplied as less than EPS, where EPS is the relative machine precision, then the value EPS is used in place of TOL.

Further Details

This subroutine is adapted from the subroutine DLAGTF in LAPACK.

6.10.38 subroutine sym_trid_cmp2 (d, e, sub, diag, sup1, sup2, perm)

Purpose

SYM_TRID_CMP2 factorizes an n by n symmetric tridiagonal matrix T, as

T = P * L * U

where P is a permutation matrix, L is a unit lower tridiagonal matrix with at most one non-zero subdiagonal elements per column and U is an upper triangular matrix with at most two non-zero superdiagonal elements per column.

The factorization is obtained by Gaussian elimination with partial pivoting and row interchanges.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the symmetric tridiagonal matrix T.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 subdiagonal elements of the symmetric tridiagonal matrix T and E(n) is arbitrary .

The size of E must be size(E) = size(D) = n .

SUB (OUTPUT) real(stnd), dimension(:) On exit, SUB(:n-1) contains the n-1 subdiagonal elements of the lower triangular matrix L of the factorization of T, SUB(n) is arbitrary .

The size of SUB must verify: size(SUB) = size(D) = n .

DIAG (OUTPUT) real(stnd), dimension(:) On exit, DIAG(:) contains the n diagonal elements of the upper triangular matrix U of the factorization of T.

The size of DIAG must verify: size(DIAG) = size(D) = n .

SUP1 (OUTPUT) real(stnd), dimension(:) On exit, SUP1(:n-1) contains the n-1 superdiagonal elements of the upper triangular matrix U of the factorization of T, SUP1(n) is arbitrary .

The size of SUP1 must verify: size(SUP1) = size(D) = n .

SUP2 (OUTPUT) real(stnd), dimension(:) On exit, SUP2(:n-2) contains the n-2 second superdiagonal elements of the upper triangular matrix U of the factorization of T, SUP2(n-1:n) is arbitrary .

The size of SUP2 must verify: size(SUP2) = size(D) = n .

PERM (OUTPUT) logical(lgl), dimension(:) On exit, PERM(:n-1) contains details of the permutation matrix P(j):

- If an interchange occurred at the kth step of the elimination in the factorization of T, then PERM(k) = TRUE
- Otherwise PERM(k) = FALSE.

PERM(n) is arbitrary .

The size of PERM must verify: size(PERM) = size(D) = n .

Further Details

SYM_TRID_CMP2 is a simplified version of SYM_TRID_CMP. This subroutine is adapted from the subroutine DGTTRF in LAPACK.

6.10.39 subroutine sym_trid_solve (sub, diag, sup1, sup2, perm, y, scale)

Purpose

SYM_TRID_SOLVE may be used to solve the system of equations

x(:) * T = scale * y(:)

, where T is an n by n symmetric tridiagonal matrix and scale is a scalar for x(:), following the factorization of T by SYM_TRID_CMP or SYM_TRID_CMP2 as

T = P * L * U

where P is a permutation matrix, L is a unit lower tridiagonal matrix with at most one non-zero subdiagonal elements per column and U is an upper triangular matrix with at most two non-zero superdiagonal elements per column.

The matrix T is assumed to be ill-conditioned, and frequent rescalings are carried out in order to avoid overflow. However, no test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

Arguments

SUB (INPUT) real(stnd), dimension(:) On entry, SUB(:n-1) contains the n-1 subdiagonal elements of the lower triangular matrix L of the factorization of T,

SUB(n) is arbitrary.

The size of SUB must verify: size(SUB) = size(Y) = n .

DIAG (INPUT) real(stnd), dimension(:) On entry, DIAG(:) contains the n diagonal elements of the upper triangular matrix U of the factorization of T.

The shape of DIAG must verify: size(DIAG) = size(Y) = n .

SUP1 (INPUT) real(stnd), dimension(:) On entry, SUP1(:n-1) contains the n-1 superdiagonal elements of the upper triangular matrix U of the factorization of T,

SUP1(n) is arbitrary.

The shape of SUP1 must verify: size(SUP1) = size(Y) = n .

SUP2 (INPUT) real(stnd), dimension(:) On entry, SUP2(:n-2) contains the n-2 second superdiagonal elements of the upper triangular matrix U of the factorization of T, SUP2(n-1:n) is arbitrary.

The shape of SUP2 must verify: size(SUP2) = size(Y) = n .

PERM (INPUT) logical(lgl), dimension(:) On entry, PERM(:n-1) contains details of the permutation matrix P:

- if an interchange occurred at the kth step of the elimination in the factorization of T, then PERM(k) = TRUE,
- otherwise PERM(k) = FALSE.

PERM(n) is arbitrary .

The shape of PERM must verify: size(PERM) = size(Y) = n .

Y (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the right hand side vector y.

On exit, Y is overwritten the solution vector x.

The shape of Y must verify: size(Y) = n .

SCALE (OUTPUT) real(stnd) On exit, the scalar SCALE.

Further Details

This subroutine is adapted from the subroutine DLAGTS in LAPACK.

6.10.40 subroutine sym_trid_solve (sub, diag, sup1, sup2, perm, y)

Purpose

SYM_TRID_SOLVE may be used to solve the system of equations

x(:) * T = y(:)

, where T is an n by n symmetric tridiagonal matrix, following the factorization of T by SYM_TRID_CMP or SYM_TRID_CMP2 as

T = P * L * U

where P is a permutation matrix, L is a unit lower tridiagonal matrix with at most one non-zero sub-diagonal elements per column and U is an upper triangular matrix with at most two non-zero super-diagonal elements per column.

The matrix T is assumed to be no singular and well-conditioned.

Arguments

SUB (INPUT) real(stnd), dimension(:) On entry, SUB(:n-1) contains the n-1 subdiagonal elements of the lower triangular matrix L of the factorization of T, SUB(n) is arbitrary .

The size of SUB must verify: size(SUB) = size(Y) = n .

DIAG (INPUT) real(stnd), dimension(:) On entry, DIAG(:) contains the n diagonal elements of the upper triangular matrix U of the factorization of T.

The shape of DIAG must verify: size(DIAG) = size(Y) = n .

SUP1 (INPUT) real(stnd), dimension(:) On entry, SUP1(:n-1) contains the n-1 superdiagonal elements of the upper triangular matrix U of the factorization of T, SUP1(n) is arbitrary.

The shape of SUP1 must verify: size(SUP1) = size(Y) = n .

SUP2 (INPUT) real(stnd), dimension(:) On entry, SUP2(:n-2) contains the n-2 second superdiagonal elements of the upper triangular matrix U of the factorization of T, SUP2(n-1:n) is arbitrary.

The shape of SUP2 must verify: size(SUP2) = size(Y) = n .

PERM (INPUT) logical(lgl), dimension(:) On entry, PERM(:n-1) contains details of the permutation matrix P:

- if an interchange occurred at the kth step of the elimination in the factorization of T, then PERM(k) = TRUE,
- otherwise PERM(k) = FALSE.

PERM(n) is arbitrary .

The shape of PERM must verify: size(PERM) = size(Y) = n .

Y (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the right hand side vector y.

On exit, Y is overwritten the solution vector x.

The shape of Y must verify: size(Y) = n .

Further Details

This subroutine is adapted from the routine DLAGTS in LAPACK.

6.11 Module_Logical_Constants

Copyright 2018 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING LOGICAL CONSTANTS OF KIND 'lgl'.

BY ONLY USING LOGICAL VALUES AS DEFINED WITHIN THIS MODULE (e.g. THE LOGICAL CONSTANTS true AND false OF KIND lgl), ALL PROBLEMS ASSOCIATED WITH THE CONVERSION OF LOGICAL LITERAL VALUES CAN BE TOTALLY AVOIDED.

LATEST REVISION : 30/05/2018

6.12 Module_Mul_Stat_Procedures

Copyright 2024 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING SUBROUTINES AND FUNCTIONS FOR MULTIVARIATE STATISTICAL COMPUTATIONS

LATEST REVISION : 12/03/2024

6.12.1 `subroutine comp_cor ( x, y, first, last, xstat, ystat, xycor, xyn, z, prob, ndf_max, cortest, cov )`

Purpose

COMP_COR computes estimates of mean, variance and correlation coefficient from two data vectors XX and YY.

Arguments

X (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X) observations from the vector of data XX for which basic univariate and bivariate statistics are desired. If all the data are available at once, X can be the full data vector XX.

Y (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X) observations from another vector of data YY for which correlation coefficient with XX is desired. If all the data are available at once, YY can be the full data vector.

The size of Y must verify: size(Y) = size(X) .

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subvector is the first subvector of the data vector XX (or YY).
- FIRST = false the current subvector is not the first subvector of the data vector XX (or YY).

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current subvector is the last subvector of the data vector XX (or YY).
- LAST = false the current subvector is not the last subvector of the data vector XX (or YY).

XSTAT (INPUT/OUTPUT) real(stnd), dimension(2) On entry, after the first call to COMP_COR (e.g., when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR. XSTAT should not be changed between calls to COMP_COR.

On exit, when LAST=true, XSTAT contains the following statistics:

- XSTAT(1) contains the mean value of the data vector XX.
- XSTAT(2) contains the variance of the data vector XX.

The size of XSTAT must verify: size(XSTAT) = 2.

YSTAT (INPUT/OUTPUT) real(stnd), dimension(2) On entry, after the first call to COMP_COR (e.g., when FIRST=true), YSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR. YSTAT should not be changed between calls to COMP_COR.

On exit, when LAST=true, YSTAT contains the following statistics:

- YSTAT(1) contains the mean value of the data vector YY.
- YSTAT(2) contains the variance of the data vector YY.

The size of YSTAT must verify: size(YSTAT) = 2.

XYCOR (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_COR (e.g., when FIRST=true), XYCOR is used as workspace to accumulate quantities on previous calls to COMP_COR. XYCOR should not be changed between calls to COMP_COR.

On exit, when LAST=true, XYCOR contains the correlation coefficient between XX(:) and YY(:).

XYN (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_COR (e.g., when FIRST=true), XYN contains count of observations from previous calls to COMP_COR. XYN should not be changed between calls to COMP_COR.

On exit, XYN contains the number of observations in the data vectors XX and YY.

Z (OUTPUT, OPTIONAL) real(stnd) On exit, when LAST=true, Z contains the Fisher's Z transformation of XYCOR.

Z needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

PROB (OUTPUT, OPTIONAL) real(stnd) On exit, when LAST=true, PROB gives the probability that the random sample of XYN observation pairs YY(:) and XX(:) came from a bivariate normal population with a correlation coefficient equal to zero.

PROB needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, when argument PROB is present, NDF_MAX is used as follows:

- If XYN-2 is lower or equal to NDF_MAX, the t_density is integrated for computing PROB.
- If XYN-2 is greater than NDF_MAX, an asymptotic series is used for computing PROB.

The default is 20.

NDF_MAX needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

CORTEST (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, a probability. CORTEST is the sum of the areas (equal) in both tails of the Student's t distribution with XYN-2 degrees of freedom.

CORTEST must verify: 0. < P < 1.

On exit, the two-tail CORTEST quantile of the sample correlation coefficient, that is a value R such that the probability of the absolute value of a sample correlation coefficient computed from XYN observation pairs being greater than R is CORTEST under the null hypothesis of no correlation in the parent normal population.

CORTEST needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

COV (INPUT, OPTIONAL) logical(lgl) On entry, if COV=true, covariance coefficient between the data vectors XX and YY is computed instead of correlation coefficient. If COV=true, Z and PROB are set to Nan code.

COV needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

Further Details

The subroutine computes the basic univariate statistics and the correlation coefficient with only one pass through the data.

If fewer than two valid observations were present, the pertinent statistics are set to Nan code.

For more details on correlation and regression analysis, see:

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 8, 484 pp., ISBN:9780521012300

6.12.2 subroutine comp_cor (x, y, first, last, xstat, ystat, xycor, xyn, dimvar, z, prob, ndf_max, cortest, cov)

Purpose

COMP_COR computes estimates of means, variances and correlation coefficients (with a data vector YY) from a data matrix XX.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data XX for which basic univariate and bivariate statistics are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix XX.

Y (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X,3-DIMVAR) observations from a vector of data YY for which correlation coefficient with XX is desired. If all the data are available at once, YY can be the full data vector.

The size of Y must verify: size(Y) = SIZE(X, 3-DIMVAR) .

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrix is the first submatrix of the data matrix XX.
- FIRST = false the current submatrix is not the first submatrix of the data matrix XX.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrix is the last submatrix of the data matrix XX.
- LAST = false the current submatrix is not the last submatrix of the data matrix XX.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,2) On entry, after the first call to COMP_COR (e.g., when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR. XSTAT should not be changed between calls to COMP_COR.

On exit, when LAST=true, XSTAT contains the following statistics on all variables:

- XSTAT(:,1) contains the mean values of the data matrix XX.
- XSTAT(:,2) contains the variances of the data matrix XX.

The shape of XSTAT must verify:

- size(XSTAT, 1) = size(X, DIMVAR),
- size(XSTAT, 2) = 2 .

YSTAT (INPUT/OUTPUT) real(stnd), dimension(2) On entry, after the first call to COMP_COR (e.g., when FIRST=true), YSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR. YSTAT should not be changed between calls to COMP_COR.

On exit, when LAST=true, YSTAT contains the following statistics:

- YSTAT(1) contains the mean value of the data vector YY.
- YSTAT(2) contains the variance of the data vector YY.

The size of YSTAT must verify: size(YSTAT) = 2.

XYCOR (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_COR (e.g., when FIRST=true), XYCOR is used as workspace to accumulate quantities on previous calls to COMP_COR. XYCOR should not be changed between calls to COMP_COR.

On exit, when LAST=true, XYCOR(i) contains the correlation coefficient between XX(i,:) (XX(:,i) if DIMVAR=2) and YY(:).

The size of XYCOR must verify: size(XYCOR) = size(X, DIMVAR).

XYN (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_COR (e.g., when FIRST=true), XYN contains count of observations from previous calls to COMP_COR. XYN should not be changed between calls to COMP_COR.

On exit, XYN contains the number of observations in the data matrix XX and the data vector YY.

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

Z (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true, Z contains the Fisher's Z transformation of XYCOR.

Z needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

The size of Z must verify: size(Z) = size(X, DIMVAR).

PROB (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true, PROB(i) gives the probability that the random sample of XYN observation pairs YY(:) and XX(i,:) (XX(:,i) if DIMVAR=2) came from a bivariate normal population with a correlation coefficient equal to zero.

PROB needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

The size of PROB must verify: size(PROB) = size(X, DIMVAR).

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, when argument PROB is present, NDF_MAX is used as follows:

- If XYN-2 is lower or equal to NDF_MAX, the t_density is integrated for computing PROB(i).
- If XYN-2 is greater than NDF_MAX, an asymptotic series is used for computing PROB(i).

The default is 20.

NDF_MAX needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

CORTEST (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, a probability. CORTEST is the sum of the areas (equal) in both tails of the Student's t distribution with XYN-2 degrees of freedom.

CORTEST must verify: 0. < P < 1.

On exit, the two-tail CORTEST quantile of the sample correlation coefficient, that is a value R such that the probability of the absolute value of a sample correlation coefficient computed from XYN observation pairs being greater than R is CORTEST under the null hypothesis of no correlation in the parent normal population.

CORTEST needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

COV (INPUT, OPTIONAL) logical(lgl) On entry, if COV=true, covariance coefficients between the data matrix XX and data vector YY are computed instead of correlation coefficients. If COV=true, Z and PROB are set to Nan code. COV needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

Further Details

The subroutine computes the basic univariate statistics and the correlation coefficients with only one pass through the data.

If fewer than two valid observations were present, the pertinent statistics are set to Nan code.

For more details on correlation and regression analysis, see:

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 8, 484 pp., ISBN:9780521012300

6.12.3 subroutine comp_cor (x, y, first, last, xstat, ystat, xycor, xyn, z, prob, ndf_max, cortest, cov)

Purpose

COMP_COR computes estimates of means, variances and correlation coefficients (with a data vector YY) from a data tridimensional array XX.

Arguments

X (INPUT) real(stnd), dimension(:,:,:) On entry, input subarray containing size(X,3) observations on size(X,1) by size(X,2) variables from the tridimensional array of data XX for which basic univariate and bivariate statistics are desired. If all the data are available at once, X can be the full data array XX.

Y (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X,3) observations from a vector of data YY for which correlation coefficients with XX is desired. If all the data are available at once, YY can be the full data vector.

The size of Y must verify: size(Y) = SIZE(X, 3) .

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subarray is the first subarray of the data array XX.
- FIRST = false the current subarray is not the first subarray of the data array XX.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current subarray is the last subarray of the data array XX.
- LAST = false the current subarray is not the last subarray of the data array XX.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,:,2) On entry, after the first call to COMP_COR (e.g., when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR. XSTAT should not be changed between calls to COMP_COR.

On exit, when LAST=true, XSTAT contains the following statistics on all variables:

- XSTAT(:,:,1) contains the mean values of the data array XX.
- XSTAT(:,:,2) contains the variances of the data array XX.

The shape of XSTAT must verify:

- size(XSTAT, 1) = size(X, 1) ,
- size(XSTAT, 2) = size(X, 2) ,
- size(XSTAT, 3) = 2 .

YSTAT (INPUT/OUTPUT) real(stnd), dimension(2) On entry, after the first call to COMP_COR (e.g., when FIRST=true), YSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR. YSTAT should not be changed between calls to COMP_COR.

On exit, when LAST=true, YSTAT contains the following statistics:

- YSTAT(1) contains the mean value of the data vector YY.
- YSTAT(2) contains the variance of the data vector YY.

The size of YSTAT must verify: size(YSTAT) = 2.

XYCOR (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_COR (e.g., when FIRST=true), XYCOR is used as workspace to accumulate quantities on previous calls to COMP_COR. XYCOR should not be changed between calls to COMP_COR.

On exit, when LAST=true, XYCOR(i,j) contains the correlation coefficient between XX(i,j,:) and YY(:).

The shape of XYCOR must verify:

- size(XYCOR, 1) = size(X, 1) ,
- size(XYCOR, 2) = size(X, 2) .

XYN (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_COR (e.g., when FIRST=true), XYN contains count of observations from previous calls to COMP_COR. XYN should not be changed between calls to COMP_COR.

On exit, XYN contains the number of observations in the data array XX and the data vector YY.

Z (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, when LAST=true, Z contains the Fisher's Z transformation of XYCOR.

Z needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

The shape of Z must verify:

- size(Z, 1) = size(X, 1) ,
- size(Z, 2) = size(X, 2) .

PROB (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, when LAST=true, PROB(i,j) gives the probability that the random sample of XYN observation pairs YY(:) and XX(i,j,:) came from a bivariate normal population with a correlation coefficient equal to zero.

PROB needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

The shape of PROB must verify:

- size(PROB, 1) = size(X, 1) ,
- size(PROB, 2) = size(X, 2) .

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, when argument PROB is present, NDF_MAX is used as follows:

- If XYN-2 is lower or equal to NDF_MAX, the t_density is integrated for computing PROB(i,j).
- If XYN-2 is greater than NDF_MAX, an asymptotic series is used for computing PROB(i,j).

The default is 20.

NDF_MAX needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

CORTEST (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, a probability. CORTEST is the sum of the areas (equal) in both tails of the Student's t distribution with XYN-2 degrees of freedom.

CORTEST must verify: 0. < P < 1.

On exit, the two-tail CORTEST quantile of the sample correlation coefficient, that is a value R such that the probability of the absolute value of a sample correlation coefficient computed from XYN observation pairs being greater than R is CORTEST under the null hypothesis of no correlation in the parent normal population.

CORTEST needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

COV (INPUT, OPTIONAL) logical(lgl) On entry, if COV=true, covariance coefficients between the data matrices XX and YY are computed instead of correlation coefficients. If COV=true, Z and PROB are set to Nan code. COV needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

Further Details

The subroutine computes the basic univariate statistics and the correlation coefficients with only one pass through the data.

If fewer than two valid observations were present, the pertinent statistics are set to Nan code.

For more details on correlation and regression analysis, see:

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 8, 484 pp., ISBN:9780521012300

6.12.4 subroutine comp_cor (x, y, first, last, xstat, ystat, xycor, xyn, dimvar, dimvary, z, prob, ndf_max, cortest, cov)

Purpose

COMP_COR computes estimates of means, variances and correlation coefficients between two data matrices YY and XX.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data XX for which basic univariate and bivariate statistics are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix XX.

Y (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(Y,3-DIMVARY) observations on size(Y,DIMVARY) variables from the matrix of data YY for which basic univariate and bivariate statistics are desired. By default, DIMVARY is equal to 1. See description of optional DIMVARY argument for details. If all the data are available at once, Y can be the full data matrix YY.

The shape of Y must verify: size(Y, 3-DIMVARY) = size(X, 3-DIMVAR) .

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrices are the first submatrices of the data matrices XX and YY.
- FIRST = false the current submatrices are not the first submatrices of the data matrices XX and YY.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrices are the last submatrices of the data matrices XX and YY.
- LAST = false the current submatrix are not the last submatrices of the data matrices XX and YY.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,2) On entry, after the first call to COMP_COR (e.g., when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR. XSTAT should not be changed between calls to COMP_COR.

On exit, when LAST=true, XSTAT contains the following statistics on all variables from the XX matrix:

- XSTAT(:,1) contains the mean values of the data matrix XX.
- XSTAT(:,2) contains the variances of the data matrix XX.

The shape of XSTAT must verify:

- size(XSTAT, 1) = size(X, DIMVAR) ,
- size(XSTAT, 2) = 2 .

YSTAT (INPUT/OUTPUT) real(stnd), dimension(:,2) On entry, after the first call to COMP_COR (e.g., when FIRST=true), YSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR. YSTAT should not be changed between calls to COMP_COR.

On exit, when LAST=true, YSTAT contains the following statistics on all variables from the YY matrix:

- YSTAT(:,1) contains the mean values of the data matrix YY.
- YSTAT(:,2) contains the variances of the data matrix YY.

The shape of YSTAT must verify:

- size(YSTAT, 1) = size(Y, DIMVARY) ,
- size(YSTAT, 2) = 2 .

XYCOR (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_COR (e.g., when FIRST=true), XYCOR is used as workspace to accumulate quantities on previous calls to COMP_COR. XYCOR should not be changed between calls to COMP_COR.

On exit, when LAST=true, XYCOR(i,j) contains the correlation coefficient between XX(i,:) and YY(j,:) (XX(:,i) and YY(:,j) if DIMVAR=2 and DIMVARY=2).

The shape of XYCOR must verify:

- size(XYCOR, 1) = size(X, DIMVAR) ,
- size(XYCOR, 2) = size(Y, DIMVARY) .

XYN (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_COR (e.g., when FIRST=true), XYN contains count of observations from previous calls to COMP_COR. XYN should not be changed between calls to COMP_COR.

On exit, XYN contains the number of observations in the data matrices XX and YY.

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables, respectively.

The default is DIMVAR = 1.

DIMVARY (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVARY is present, DIMVARY is used as follows:

- DIMVARY = 1, the input submatrix Y contains size(Y,2) observations on size(Y,1) variables.
- DIMVARY = 2, the input submatrix Y contains size(Y,1) observations on size(Y,2) variables, respectively.

The default is DIMVARY = 1.

Z (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, when LAST=true, Z contains the Fisher's Z transformation of XYCOR.

Z needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

The shape of Z must verify:

- size(Z, 1) = size(X, DIMVAR) ,
- size(Z, 2) = size(Y, DIMVARY) .

PROB (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, when LAST=true, PROB(i,j) gives the probability that the random sample of XYN observation pairs XX(i,:) and YY(j,:) (XX(:,i) and YY(:,j) if DIMVAR=2 and DIMVARY=2) came from a bivariate normal population with a correlation coefficient equal to zero.

PROB needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

The shape of PROB must verify:

- size(PROB, 1) = size(X, DIMVAR) ,
- size(PROB, 2) = size(Y, DIMVARY) .

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, when argument PROB is present, NDF_MAX is used as follows:

- If XYN-2 is lower or equal to NDF_MAX, the t_density is integrated for computing PROB(i,j).
- If XYN-2 is greater than NDF_MAX, an asymptotic series is used for computing PROB(i,j).

The default is 20.

NDF_MAX needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

CORTEST (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, a probability. CORTEST is the sum of the areas (equal) in both tails of the Student's t distribution with XYN-2 degrees of freedom.

CORTEST must verify: 0. < P < 1.

On exit, the two-tail CORTEST quantile of the sample correlation coefficient, that is a value R such that the probability of the absolute value of a sample correlation coefficient computed from XYN observation pairs being greater than R is CORTEST under the null hypothesis of no correlation in the parent normal population.

CORTEST needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

COV (INPUT, OPTIONAL) logical(lgl) On entry, if COV=true, covariance coefficients between the data matrices XX and YY are computed instead of correlation coefficients. If COV=true, Z and PROB are set to Nan code.

COV needs to be specified only on the last call to COMP_COR (e.g., when LAST=true).

Further Details

The subroutine computes the basic univariate statistics and the correlation coefficients with only one pass through the data.

If fewer than two valid observations were present, the pertinent statistics are set to Nan code.

For more details on correlation and regression analysis, see:

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 8, 484 pp., ISBN:9780521012300

6.12.5 `subroutine comp_cor_miss ( x, y, first, last, xstat, ystat, xycor, xymiss, z, prob, ndf_max, cov )`

Purpose

COMP_COR_MISS computes estimates of mean, variance and correlation coefficient from two data vectors XX and YY possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X) observations from the vector of data XX for which basic univariate and bivariate statistics are desired. If all the data are available at once, X can be the full data vector XX.

Y (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X) observations from another vector of data YY for which correlation coefficient with XX is desired. If all the data are available at once, YY can be the full data vector.

The size of Y must verify: size(Y) = size(X) .

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subvector is the first subvector of the data vector XX (or YY).
- FIRST = false the current subvector is not the first subvector of the data vector XX (or YY).

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current subvector is the last subvector of the data vector XX (or YY).
- LAST = false the current subvector is not the last subvector of the data vector XX (or YY).

XSTAT (INPUT/OUTPUT) real(stnd), dimension(4) On entry, after the first call to COMP_COR_MISS (e.g., when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS. XSTAT should not be changed between calls to COMP_COR_MISS.

On exit, when LAST=true, XSTAT contains the following statistics:

- XSTAT(1) contains the mean value of the data vector XX.
- XSTAT(2) contains the variance of the data vector XX.
- XSTAT(3) contains the the number of non-missing observations in the data vector XX.
- XSTAT(4) is used as workspace.

The size of XSTAT must verify: size(XSTAT) = 4.

YSTAT (INPUT/OUTPUT) real(stnd), dimension(4) On entry, after the first call to COMP_COR_MISS (e.g., when FIRST=true), YSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS. YSTAT should not be changed between calls to COMP_COR_MISS.

On exit, when LAST=true, YSTAT contains the following statistics:

- YSTAT(1) contains the mean value of the data vector YY.
- YSTAT(2) contains the variance of the data vector YY.
- YSTAT(3) contains the the number of non-missing observations in the data vector YY.
- YSTAT(4) is used as workspace.

The size of YSTAT must verify: size(YSTAT) = 4.

XYCOR (INPUT/OUTPUT) real(stnd), dimension(4) On entry, after the first call to COMP_COR_MISS (e.g., when FIRST=true), XYCOR is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS. XYCOR should not be changed between calls to COMP_COR_MISS.

On exit, when LAST=true, XYCOR contains the following statistics:

- XYCOR(1) contains the correlation coefficient between XX(:) and YY(:).

- XYCOR(2) contains the incidence value between XX(:) and YY(:). XYCOR(2) indicates the number of non-missing pairs of observations which were used in the calculation of XYCOR(1).
- XYCOR(3:4) is used as workspace.

The size of XYCOR must verify: size(XYCOR) = 4.

XYMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X or Y which is equal to XYMISS is assumed to be missing or invalid. The basic univariate and bivariate statistics are computed on all the observations where XX and YY are not missing (see Further Details).

Z (OUTPUT, OPTIONAL) real(stnd) On exit, when LAST=true, Z contains the Fisher's Z transformation of XYCOR(1).

Z needs to be specified only on the last call to COMP_COR_MISS (e.g., when LAST=true).

PROB (OUTPUT, OPTIONAL) real(stnd) On exit, when LAST=true, PROB gives the probability that the random sample of XYCOR(2) observation pairs YY(:) and XX(:) came from a bivariate normal population with a correlation coefficient equal to zero.

PROB needs to be specified only on the last call to COMP_COR_MISS (e.g., when LAST=true).

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, when argument PROB is present, NDF_MAX is used as follows, if:

- XYCOR(2)-2 is lower or equal to NDF_MAX, the t_density is integrated for computing PROB.
- XYCOR(2)-2 is greater than NDF_MAX, an asymptotic series is used for computing PROB.

The default is 20.

NDF_MAX needs to be specified only on the last call to COMP_COR_MISS (e.g., when LAST=true).

COV (INPUT, OPTIONAL) logical(lgl) On entry, if COV=true, covariance coefficient between the data vectors XX and YY are computed instead of correlation coefficient. If COV=true, Z and PROB are set to XYMISS .

COV needs to be specified only on the last call to COMP_COR_MISS (e.g., when LAST=true).

Further Details

The subroutine computes the basic univariate statistics and the correlation coefficient with only one pass through the data.

If fewer than two valid observations were present, the pertinent statistics are set to XYMISS.

The means and standard-deviations of XX and YY are computed from all valid data. The correlation coefficients are based on these univariate statistics and on all valid pairs of observations.

For more details on correlation and regression analysis, see:

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 8, 484 pp., ISBN:9780521012300

6.12.6 `subroutine comp_cor_miss ( x, y, first, last, xstat, ystat, xycor, xymiss, dimvar, z, prob, ndf_max, cov )`

Purpose

COMP_COR_MISS computes estimates of means, variances and correlation coefficients (with another data vector YY) from a data matrix XX possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data XX for which basic univariate and bivariate statistics are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix XX.

Y (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X,3-DIMVAR) observations from a vector of data YY for which correlation coefficient with XX is desired. If all the data are available at once, YY can be the full data vector.

The size of Y must verify: size(Y) = SIZE(X, 3-DIMVAR) .

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrix is the first submatrix of the data matrix XX.
- FIRST = false the current submatrix is not the first submatrix of the data matrix XX.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrix is the last submatrix of the data matrix XX.
- LAST = false the current submatrix is not the last submatrix of the data matrix XX.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,4) On entry, after the first call to COMP_COR_MISS (e.g., when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS. XSTAT should not be changed between calls to COMP_COR_MISS.

On exit, when LAST=true, XSTAT contains the following statistics on all variables:

- XSTAT(:,1) contains the mean values of the data matrix XX.
- XSTAT(:,2) contains the variances of the data matrix XX.
- XSTAT(:,3) contains the the numbers of non-missing observations in the data matrix XX.
- XSTAT(:,4) is used as workspace.

The shape of XSTAT must verify:

- size(XSTAT, 1) = size(X, DIMVAR) ,
- size(XSTAT, 2) = 4 .

YSTAT (INPUT/OUTPUT) real(stnd), dimension(4) On entry, after the first call to COMP_COR_MISS (e.g., when FIRST=true), YSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS. YSTAT should not be changed between calls to COMP_COR_MISS.

On exit, when LAST=true, YSTAT contains the following statistics:

- YSTAT(1) contains the mean value of the data vector YY.
- YSTAT(2) contains the variance of the data vector YY.
- YSTAT(3) contains the the number of non-missing observations in the data vector YY.

- YSTAT(4) is used as workspace.

The size of YSTAT must verify: size(YSTAT) = 4.

XYCOR (INPUT/OUTPUT) real(stnd), dimension(:,4) On entry, after the first call to COMP_COR_MISS (e.g., when FIRST=true), XYCOR is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS. XYCOR should not be changed between calls to COMP_COR_MISS.

On exit, when LAST=true, XYCOR contains the following statistics:

- XYCOR(i,1) contains the correlation coefficients between XX(i,:) (XX(:,i) if DIMVAR=2) and YY(:).
- XYCOR(i,2) contains the incidence values between XX(i,:) (XX(:,i) if DIMVAR=2) and YY(:). XYCOR(i,2) indicates the numbers of non-missing pairs of observations which were used in the calculation of XYCOR(i,1).
- XYCOR(:,3:4) is used as workspace.

The shape of XYCOR must verify:

- size(XYCOR, 1) = size(X, DIMVAR) ,
- size(XYCOR, 2) = 4 .

XYMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X or Y which is equal to XYMISS is assumed to be missing or invalid. The basic univariate and bivariate statistics are computed on all the observations where X and Y are not missing (see Further Details).

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

Z (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true, Z contains the Fisher's Z transformation of XYCOR(:,1).

Z needs to be specified only on the last call to COMP_COR_MISS (e.g., when LAST=true).

The size of Z must verify: size(Z) = size(X, DIMVAR).

PROB (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true, PROB(i) gives the probability that the random sample of XYCOR(i,2) observation pairs YY(:) and XX(i,:) (XX(:,i) if DIMVAR=2) came from a bivariate normal population with a correlation coefficient equal to zero.

PROB needs to be specified only on the last call to COMP_COR_MISS (e.g., when LAST=true).

The size of PROB must verify: size(PROB) = size(X, DIMVAR).

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, when argument PROB is present, NDF_MAX is used as follows:

- If XYCOR(i,2)-2 is lower or equal to NDF_MAX, the t_density is integrated for computing PROB(i).
- If XYCOR(i,2)-2 is greater than NDF_MAX, an asymptotic series is used for computing PROB(i).

The default is 20.

NDF_MAX needs to be specified only on the last call to COMP_COR_MISS (e.g., when LAST=true).

COV (INPUT, OPTIONAL) logical(lgl) On entry, if COV=true, covariance coefficients between the data matrix XX and data vector YY are computed instead of correlation coefficients. If COV=true, Z and PROB are set to XYMISS .

COV needs to be specified only on the last call to COMP_COR_MISS (e.g., when LAST=true).

Further Details

The subroutine computes the basic univariate statistics and the correlation coefficients with only one pass through the data.

If fewer than two valid observations were present, the statistics are set to XYMISS .

The means and standard-deviations of XX and YY are computed from all valid data. The correlation coefficients are based on these univariate statistics and on all valid pairs of observations.

For more details on correlation and regression analysis, see:

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 8, 484 pp., ISBN:9780521012300

6.12.7 `subroutine comp_cor_miss ( x, y, first, last, xstat, ystat, xycor, xymiss, z, prob, ndf_max, cov )`

Purpose

COMP_COR_MISS computes estimates of means, variances and correlation coefficients (with another data vector YY) from a data tridimensional array XX possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:,:) On entry, input subarray containing size(X,3) observations on size(X,1) by size(X,2) variables from the array of data XX for which basic univariate and bivariate statistics are desired. If all the data are available at once, X can be the full data array XX.

Y (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X,3) observations from a vector of data YY for which correlation coefficient with XX is desired. If all the data are available at once, YY can be the full data vector.

The size of Y must verify: size(Y) = SIZE(X, 3) .

FIRST (INPUT) logical(lgl) On entry, if

- FIRST = true the current subarray is the first subarray of the data array XX.
- FIRST = false the current subarray is not the first subarray of the data array XX.

LAST (INPUT) logical(lgl) On entry, if

- LAST = true the current subarray is the last subarray of the data array XX.
- LAST = false the current subarray is not the last subarray of the data array XX.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,:,4) On entry, after the first call to COMP_COR_MISS (e.g., when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS. XSTAT should not be changed between calls to COMP_COR_MISS.

On exit, when LAST=true, XSTAT contains the following statistics on all variables:

- XSTAT(:,:,1) contains the mean values of the data array XX.
- XSTAT(:,:,2) contains the variances of the data array XX.
- XSTAT(:,:,3) contains the numbers of non-missing observations in the data array XX.
- XSTAT(:,:,4) is used as workspace.

The shape of XSTAT must verify:

- size(XSTAT, 1) = size(X, 1) ,
- size(XSTAT, 2) = size(X, 2) ,
- size(XSTAT, 3) = 4 .

YSTAT (INPUT/OUTPUT) real(stnd), dimension(4) On entry, after the first call to COMP_COR_MISS (e.g., when FIRST=true), YSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS. YSTAT should not be changed between calls to COMP_COR_MISS.

On exit, when LAST=true, YSTAT contains the following statistics:

- YSTAT(1) contains the mean value of the data vector YY.
- YSTAT(2) contains the variance of the data vector YY.
- YSTAT(3) contains the the number of non-missing observations in the data vector YY.
- YSTAT(4) is used as workspace.

The size of YSTAT must verify: size(YSTAT) = 4.

XYCOR (INPUT/OUTPUT) real(stnd), dimension(:,:,4) On entry, after the first call to COMP_COR_MISS (e.g., when FIRST=true), XYCOR is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS. XYCOR should not be changed between calls to COMP_COR_MISS.

On exit, when LAST=true, XYCOR contains the following statistics:

- XYCOR(i,j,1) contains the correlation coefficients between XX(i,j,:) and YY(:).
- XYCOR(i,j,2) contains the incidence values between XX(i,j,:) and YY(:). XYCOR(i,j,2) indicates the numbers of valid pairs of observations which were used in the calculation of XYCOR(:,:,1).
- XYCOR(:,:,3:4) is used as workspace.

The shape of XYCOR must verify:

- size(XYCOR, 1) = size(X, 1) ,
- size(XYCOR, 2) = size(X, 2) ,
- size(XYCOR, 3) = 4 .

XYMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X or Y which is equal to XYMISS is assumed to be missing or invalid. The basic univariate and bivariate statistics are computed on all the observations where X and Y are not missing (see Further Details).

Z (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, when LAST=true, Z contains the Fisher's Z transformation of XYCOR(:,:,1).

Z needs to be specified only on the last call to COMP_COR_MISS (e.g., when LAST=true).

The shape of Z must verify:

- size(Z, 1) = size(X, 1) ,
- size(Z, 2) = size(X, 2) .

PROB (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, when LAST=true, PROB(i,j) gives the probability that the random sample of XYCOR(i,j,2) observation pairs YY(:) and XX(i,j,:) came from a bivariate normal population with a correlation coefficient equal to zero.

PROB needs to be specified only on the last call to COMP_COR_MISS (e.g., when LAST=true).

The shape of PROB must verify:

- size(PROB, 1) = size(X, 1) ,
- size(PROB, 2) = size(X, 2) .

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, when argument PROB is present, NDF_MAX is used as follows:

- If XYCOR(i,j,2)-2 is lower or equal to NDF_MAX, the t_density is integrated for computing PROB(i,j).
- If XYCOR(i,j,2)-2 is greater than NDF_MAX, an asymptotic series is used for computing PROB(i,j).

The default is 20.

NDF_MAX needs to be specified only on the last call to COMP_COR_MISS (e.g., when LAST=true).

COV (INPUT, OPTIONAL) logical(lgl) On entry, if COV=true, covariance coefficients between the data array XX and the data vector YY are computed instead of correlation coefficients. If COV=true, Z and PROB are set to XYMISS .

COV needs to be specified only on the last call to COMP_COR_MISS (e.g., when LAST=true).

Further Details

The subroutine computes the basic univariate statistics and the correlation coefficients with only one pass through the data.

If fewer than two valid observations were present, the statistics are set to XYMISS .

The means and standard-deviations of XX and YY are computed from all valid data. The correlation coefficients are based on these univariate statistics and on all valid pairs of observations.

For more details on correlation and regression analysis, see:

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 8, 484 pp., ISBN:9780521012300

6.12.8 subroutine comp_cor_miss (x, y, first, last, xstat, ystat, xycor, xymiss, dimvar, dimvary, z, prob, ndf_max, cov)

Purpose

COMP_COR_MISS computes estimates of means, variances and correlation coefficients between two data matrices YY and XX possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data XX for which basic univariate and bivariate statistics are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix XX.

Y (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(Y,3-DIMVARY) observations on size(Y,DIMVARY) variables from the matrix of data YY for which basic univariate and bivariate statistics are desired. By default, DIMVARY is equal to 1. See description of optional DIMVARY argument for details. If all the data are available at once, Y can be the full data matrix YY.

The shape of Y must verify: size(Y, 3-DIMVARY) = size(X, 3-DIMVAR) .

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrices are the first submatrices of the data matrices XX and YY.
- FIRST = false the current submatrices are not the first submatrices of the data matrices XX and YY.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrices are the last submatrices of the data matrices XX and YY.
- LAST = false the current submatrix are not the last submatrices of the data matrices XX and YY.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,4) On entry, after the first call to COMP_COR_MISS (e.g., when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS. XSTAT should not be changed between calls to COMP_COR_MISS.

On exit, when LAST=true, XSTAT contains the following statistics on all variables from the XX matrix:

- XSTAT(:,1) contains the mean values of the data matrix XX.
- XSTAT(:,2) contains the variances of the data matrix XX.
- XSTAT(:,3) contains the the numbers of non-missing observations in the data matrix XX.
- XSTAT(:,4) is used as workspace.

The shape of XSTAT must verify:

- size(XSTAT, 1) = size(X, DIMVAR) ,
- size(XSTAT, 2) = 4 .

YSTAT (INPUT/OUTPUT) real(stnd), dimension(:,4) On entry, after the first call to COMP_COR_MISS (e.g., when FIRST=true), YSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS. YSTAT should not be changed between calls to COMP_COR_MISS.

On exit, when LAST=true, YSTAT contains the following statistics on all variables from the YY matrix:

- YSTAT(:,1) contains the mean values of the data matrix YY.
- YSTAT(:,2) contains the variances of the data matrix YY.
- YSTAT(:,3) contains the the numbers of non-missing observations in the data matrix YY.
- YSTAT(:,4) is used as workspace.

The shape of YSTAT must verify:

- size(YSTAT, 1) = size(Y, DIMVARY) ,
- size(YSTAT, 2) = 4 .

XYCOR (INPUT/OUTPUT) real(stnd), dimension(:,:,4) On entry, after the first call to COMP_COR_MISS (e.g., when FIRST=true), XYCOR is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS. XYCOR should not be changed between calls to COMP_COR_MISS.

On exit, when LAST=true, XYCOR contains the following statistics:

- XYCOR(i,j,1) contains the correlation coefficients between XX(i,:) and YY(j,:) (XX(:,i) and YY(:,j) if DIMVAR=2 and DIMVARY=2).
- XYCOR(i,j,2) contains the incidence values between XX(i,:) and YY(j,:) (XX(:,i) and YY(:,j) if DIMVAR=2 and DIMVARY=2). XYCOR(i,j,2) indicates the numbers of non-missing pairs of observations which were used in the calculation of XYCOR(i,j,1).
- XYCOR(:,:,3:4) is used as workspace.

The shape of XYCOR must verify:

- size(XYCOR, 1) = size(X, DIMVAR) ,
- size(XYCOR, 2) = size(Y, DIMVARY) ,
- size(XYCOR, 3) = 4 .

XYMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X or Y which is equal to XYMISS is assumed to be missing or invalid. The basic univariate and bivariate statistics are computed on all the observations where X and Y are not missing (see Further Details).

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables, respectively.

The default is DIMVAR = 1.

DIMVARY (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVARY is present, DIMVARY is used as follows:

- DIMVARY = 1, the input submatrix Y contains size(Y,2) observations on size(Y,1) variables.
- DIMVARY = 2, the input submatrix Y contains size(Y,1) observations on size(Y,2) variables, respectively.

The default is DIMVARY = 1.

Z (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, when LAST=true, Z contains the Fisher's Z transformation of XYCOR.

Z needs to be specified only on the last call to COMP_COR_MISS (e.g., when LAST=true).

The shape of Z must verify:

- size(Z, 1) = size(X, DIMVAR) ,
- size(Z, 2) = size(Y, DIMVARY) .

PROB (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, when LAST=true, PROB(i,j) gives the probability that the random sample of XYCOR(i,j,2) observation pairs XX(i,:) and YY(j,:) (XX(:,i) and YY(:,j) if DIMVAR=2 and DIMVARY=2) came from a bivariate normal population with a correlation coefficient equal to zero.

PROB needs to be specified only on the last call to COMP_COR_MISS (e.g., when LAST=true).

The shape of PROB must verify:

- size(PROB, 1) = size(X, DIMVAR) ,
- size(PROB, 2) = size(Y, DIMVARY) .

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, when argument PROB is present, NDF_MAX is used as follows:

- If XYCOR(i,j,2)-2 is lower or equal to NDF_MAX, the t_density is integrated for computing PROB(i,j).
- If XYCOR(i,j,2)-2 is greater than NDF_MAX, an asymptotic series is used for computing PROB(i,j).

The default is 20.

NDF_MAX needs to be specified only on the last call to COMP_COR_MISS (e.g., when LAST=true).

COV (INPUT, OPTIONAL) logical(lgl) On entry, if COV=true, covariance coefficients between the data matrices XX and YY are computed instead of correlation coefficients. If COV=true, Z and PROB are set to XYMISS .

COV needs to be specified only on the last call to COMP_COR_MISS (e.g., when LAST=true).

Further Details

The subroutine computes the basic univariate statistics and the correlation coefficients with only one pass through the data.

If fewer than two valid observations were present, the statistics are set to XYMISS .

The means and standard-deviations of XX and YY are computed from all valid data. The correlation coefficients are based on these univariate statistics and on all valid pairs of observations.

For more details on correlation and regression analysis, see

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 8, 484 pp., ISBN:9780521012300

6.12.9 subroutine comp_cor_miss2 (x, y, first, last, xstat, ystat, xycor, xyn, xymiss, z, prob, ndf_max)

Purpose

COMP_COR_MISS2 computes estimates of mean, variance and correlation coefficient from two data vectors XX and YY possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X) observations from the vector of data XX for which basic univariate and bivariate statistics are desired. If all the data are available at once, X can be the full data vector XX.

Y (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X) observations from another vector of data YY for which correlation coefficient with XX is desired. If all the data are available at once, YY can be the full data vector.

The size of Y must verify: size(Y) = size(X) .

FIRST (INPUT) logical(lgl) On entry, if

- FIRST = true the current subvector is the first subvector of the data vector XX (or YY).
- FIRST = false the current subvector is not the first subvector of the data vector XX (or YY).

LAST (INPUT) logical(lgl) On entry, if

- LAST = true the current subvector is the last subvector of the data vector XX (or YY).
- LAST = false the current subvector is not the last subvector of the data vector XX (or YY).

XSTAT (INPUT/OUTPUT) real(stnd), dimension(2) On entry, after the first call to COMP_COR_MISS2 (e.g., when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS2. XSTAT should not be changed between calls to COMP_COR_MISS2.

On exit, when LAST=true, XSTAT contains the following statistics:

- XSTAT(1) contains the mean value of the data vector XX.
- XSTAT(2) contains the variance of the data vector XX.

The size of XSTAT must verify: size(XSTAT) = 2.

YSTAT (INPUT/OUTPUT) real(stnd), dimension(2) On entry, after the first call to COMP_COR_MISS2 (e.g., when FIRST=true), YSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS2. YSTAT should not be changed between calls to COMP_COR_MISS2.

On exit, when LAST=true, YSTAT contains the following statistics:

- YSTAT(1) contains the mean value of the data vector YY.
- YSTAT(2) contains the variance of the data vector YY.

The size of YSTAT must verify: size(YSTAT) = 2.

XYCOR (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_COR_MISS2 (e.g., when FIRST=true), XYCOR is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS2. XYCOR should not be changed between calls to COMP_COR_MISS2.

On exit, when LAST=true, XYCOR contains the correlation coefficient between XX(:) and YY(:).

XYN (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_COR_MISS2 (e.g., when FIRST=true), XYN contains count of valid pairs of observations from previous calls to COMP_COR_MISS2. XYN should not be changed between calls to COMP_COR_MISS2.

On exit, XYN contains the incidence value between XX(:) and YY(:). XYN indicates the number of non-missing pairs of observations which were used in the calculation of XSTAT, YSTAT and XYCOR.

XYMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X or Y which is equal to XYMISS is assumed to be missing or invalid. The basic univariate and bivariate statistics are computed on all valid pairs of observations where XX and YY are not missing (see Further Details).

Z (OUTPUT, OPTIONAL) real(stnd) On exit, when LAST=true, Z contains the Fisher's Z transformation of XYCOR.

Z needs to be specified only on the last call to COMP_COR_MISS2 (e.g., when LAST=true).

PROB (OUTPUT, OPTIONAL) real(stnd) On exit, when LAST=true, PROB gives the probability that the random sample of XYN observation pairs YY(:) and XX(:) came from a bivariate normal population with a correlation coefficient equal to zero.

PROB needs to be specified only on the last call to COMP_COR_MISS2 (e.g., when LAST=true).

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, when argument PROB is present, NDF_MAX is used as follows:

- If XYN-2 is lower or equal to NDF_MAX, the t_density is integrated for computing PROB.
- If XYN-2 is greater than NDF_MAX, an asymptotic series is used for computing PROB.

The default is 20.

NDF_MAX needs to be specified only on the last call to COMP_COR_MISS2 (e.g., when LAST=true).

Further Details

The subroutine computes the basic univariate statistics and the correlation coefficient with only one pass through the data.

If fewer than two valid observations were present, the pertinent statistics are set to XYMISS.

The univariate and bivariate statistics are computed from all valid pairs of observations.

For more details on correlation and regression analysis, see:

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 8, 484 pp., ISBN:9780521012300

6.12.10 subroutine comp_cor_miss2 (x, y, first, last, xstat, ystat, xycor, xyn, xymiss, dimvar, z, prob, ndf_max)

Purpose

COMP_COR_MISS2 computes estimates of means, variances and correlation coefficients (with another data vector YY) from a data matrix XX possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data XX for which basic univariate and bivariate statistics are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix XX.

Y (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X,3-DIMVAR) observations from a vector of data YY for which correlation coefficient with XX is desired. If all the data are available at once, YY can be the full data vector.

The size of Y must verify: size(Y) = SIZE(X, 3-DIMVAR) .

FIRST (INPUT) logical(lgl) On entry, if

- FIRST = true the current submatrix is the first submatrix of the data matrix XX.
- FIRST = false the current submatrix is not the first submatrix of the data matrix XX.

LAST (INPUT) logical(lgl) On entry, if

- LAST = true the current submatrix is the last submatrix of the data matrix XX.
- LAST = false the current submatrix is not the last submatrix of the data matrix XX.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,2) On entry, after the first call to COMP_COR_MISS2 (e.g., when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS2. XSTAT should not be changed between calls to COMP_COR_MISS2.

On exit, when LAST=true, XSTAT contains the following statistics on all variables:

- XSTAT(:,1) contains the mean values of the data matrix XX.
- XSTAT(:,2) contains the variances of the data matrix XX.

The shape of XSTAT must verify:

- size(XSTAT, 1) = size(X, DIMVAR) ,
- size(XSTAT, 2) = 2.

YSTAT (INPUT/OUTPUT) real(stnd), dimension(:,2) On entry, after the first call to COMP_COR_MISS2 (e.g., when FIRST=true), YSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS2. YSTAT should not be changed between calls to COMP_COR_MISS2.

The shape of YSTAT must verify:

- size(YSTAT, 1) = size(X, DIMVAR) ,
- size(YSTAT, 2) = 2.

XYCOR (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_COR_MISS2 (e.g., when FIRST=true), XYCOR is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS2. XYCOR should not be changed between calls to COMP_COR_MISS2.

On exit, when LAST=true, XYCOR(i) contains the correlation coefficient between XX(i,:) (XX(:,i) if DIMVAR=2) and YY(:).

The size of XYCOR must verify: size(XYCOR) = size(X, DIMVAR).

XYN (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_COR_MISS2 (e.g., when FIRST=true), XYN contains counts of valid pairs of observations from previous calls to COMP_COR_MISS2. XYN should not be changed between calls to COMP_COR_MISS2.

On exit, XYN(i) contains the incidence value between XX(i,:) (XX(:,i) if DIMVAR=2) and YY(:). XYN(i) indicates the number of non-missing pairs of observations which were used in the calculation of XSTAT and XYCOR.

The size of XYN must verify: size(XYN) = size(X, DIMVAR).

XYMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X or Y which is equal to XYMISS is assumed to be missing or invalid. The basic univariate and bivariate statistics for variable XX(i,:) (XX(:,i) if DIMVAR=2) are computed on all valid pairs of observations where XX(i,:) (XX(:,i) if DIMVAR=2) and YY(:) are not missing (see Further Details).

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

Z (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true, Z contains the Fisher's Z transformation of XYCOR(:).

Z needs to be specified only on the last call to COMP_COR_MISS2 (e.g., when LAST=true).

The size of Z must verify: size(Z) = size(X, DIMVAR).

PROB (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true, PROB(i) gives the probability that the random sample of XYN(i) observation pairs YY(:) and XX(i,:) (XX(:,i) if DIMVAR=2) came from a bivariate normal population with a correlation coefficient equal to zero.

PROB needs to be specified only on the last call to COMP_COR_MISS2 (e.g., when LAST=true).

The size of PROB must verify: size(PROB) = size(X, DIMVAR).

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, when argument PROB is present, NDF_MAX is used as follows:

- If XYN(i)-2 is lower or equal to NDF_MAX, the t_density is integrated for computing PROB(i).
- If XYN(i)-2 is greater than NDF_MAX, an asymptotic series is used for computing PROB(i).

The default is 20.

NDF_MAX needs to be specified only on the last call to COMP_COR_MISS2 (e.g., when LAST=true).

Further Details

The subroutine computes the basic univariate statistics and the correlation coefficients with only one pass through the data.

If fewer than two valid observations were present, the statistics are set to XYMISS .

The univariate and bivariate statistics are computed from all valid pairs of observations.

For more details on correlation and regression analysis, see:

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 8, 484 pp., ISBN:9780521012300

6.12.11 subroutine comp_cor_miss2 (x, y, first, last, xstat, ystat, xycor, xyn, xymiss, z, prob, ndf_max)

Purpose

COMP_COR_MISS2 computes estimates of means, variances and correlation coefficients (with another data vector YY) from a data tridimensional array XX possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:,:) On entry, input subarray containing size(X,3) observations on size(X,1) by size(X,2) variables from the array of data XX for which basic univariate and bivariate statistics are desired. If all the data are available at once, X can be the full data array XX.

Y (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X,3) observations from a vector of data YY for which correlation coefficient with XX is desired. If all the data are available at once, YY can be the full data vector.

The size of Y must verify: size(Y) = SIZE(X, 3) .

FIRST (INPUT) logical(lgl) On entry, if

- FIRST = true the current subarray is the first subarray of the data array XX.
- FIRST = false the current subarray is not the first subarray of the data array XX.

LAST (INPUT) logical(lgl) On entry, if

- LAST = true the current subarray is the last subarray of the data array XX.
- LAST = false the current subarray is not the last subarray of the data array XX.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,:,2) On entry, after the first call to COMP_COR_MISS2 (e.g., when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS2. XSTAT should not be changed between calls to COMP_COR_MISS2.

On exit, when LAST=true, XSTAT contains the following statistics on all variables:

- XSTAT(:,:,1) contains the mean values of the data array XX.
- XSTAT(:,:,2) contains the variances of the data array XX.

The shape of XSTAT must verify:

- size(XSTAT, 1) = size(X, 1) ,
- size(XSTAT, 2) = size(X, 2) ,
- size(XSTAT, 3) = 2 .

YSTAT (INPUT/OUTPUT) real(stnd), dimension(:,:,2) On entry, after the first call to COMP_COR_MISS2 (e.g., when FIRST=true), YSTAT is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS2. YSTAT should not be changed between calls to COMP_COR_MISS2.

The shape of XSTAT must verify:

- size(YSTAT, 1) = size(X, 1) ,
- size(YSTAT, 2) = size(X, 2) ,
- size(YSTAT, 3) = 2.

XYCOR (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_COR_MISS2 (e.g., when FIRST=true), XYCOR is used as workspace to accumulate quantities on previous calls to COMP_COR_MISS2. XYCOR should not be changed between calls to COMP_COR_MISS2.

On exit, when LAST=true, XYCOR(i,j) contains the correlation coefficient between XX(i,j,:) and YY(:).

The shape of XYCOR must verify:

- size(XYCOR, 1) = size(X, 1) ,
- size(XYCOR, 2) = size(X, 2) .

XYN (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_COR_MISS2 (e.g., when FIRST=true), XYN contains counts of valid pairs of observations from previous calls to COMP_COR_MISS2. XYN should not be changed between calls to COMP_COR_MISS2. On exit, XYN(i,j) contains the incidence value between XX(i,j,:) and YY(:). XYN(i,j) indicates the number of non-missing pairs of observations which were used in the calculation of XSTAT and XYCOR.

The shape of XYN must verify:

- size(XYN, 1) = size(X, 1) ,
- size(XYN, 2) = size(X, 2) .

XYMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X or Y which is equal to XYMISS is assumed to be missing or invalid. The basic univariate and bivariate statistics for variable XX(i,j,:) are computed on all valid pairs of observations where XX(i,j,:) and YY(:) are not missing (see Further Details).

Z (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, when LAST=true, Z contains the Fisher's Z transformation of XYCOR(:,:).

Z needs to be specified only on the last call to COMP_COR_MISS2 (e.g., when LAST=true).

The shape of Z must verify:

- size(Z, 1) = size(X, 1) ,
- size(Z, 2) = size(X, 2) .

PROB (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, when LAST=true, PROB(i,j) gives the probability that the random sample of XYN(i,j) observation pairs YY(:) and XX(i,j,:) came from a bivariate normal population with a correlation coefficient equal to zero.

PROB needs to be specified only on the last call to COMP_COR_MISS2 (e.g., when LAST=true).

The shape of PROB must verify:

- size(PROB, 1) = size(X, 1) ,
- size(PROB, 2) = size(X, 2) .

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, when argument PROB is present, NDF_MAX is used as follows:

- If XYN(i,j)-2 is lower or equal to NDF_MAX, the t_density is integrated for computing PROB(i,j).
- If XYN(i,j)-2 is greater than NDF_MAX, an asymptotic series is used for computing PROB(i,j).

The default is 20.

NDF_MAX needs to be specified only on the last call to COMP_COR_MISS2 (e.g., when LAST=true).

Further Details

The subroutine computes the basic univariate statistics and the correlation coefficients with only one pass through the data.

If fewer than two valid observations were present, the statistics are set to XYMISS .

The univariate and bivariate statistics are computed from all valid pairs of observations.

For more details on correlation and regression analysis, see:

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 8, 484 pp., ISBN:9780521012300

6.12.12 `subroutine update_cor ( xstat, ystat, xycor, xyn, xstat2, ystat2, xycor2, xyn2 )`

Purpose

UPDATE_COR computes sample means and corrected sums of squares and cross-products for a sample of size XYN+XYN2 given the means and corrected sum of squares and cross-products for two subsamples of size XYN and XYN2 as output by a call to COMP_COR when LAST=false on the two subsamples separetely.

The sample means, variances and coefficient correlation for the sample of size XYN+XYN2 may be obtained by a call to COMP_COR with LAST=true.

Arguments

XSTAT (INPUT/OUTPUT) real(stnd), dimension(2) On entry, the XSTAT argument of COMP_COR for the first subsample. On exit, the XSTAT argument of the combined sample.

YSTAT (INPUT/OUTPUT) real(stnd), dimension(2) On entry, the YSTAT argument of COMP_COR for the first subsample. On exit, the YSTAT argument of the combined sample.

XYCOR (INPUT/OUTPUT) real(stnd) On entry, the XYCOR argument of COMP_COR for the first subsample. On exit, the XYCOR argument of the combined sample.

XYN (INPUT/OUTPUT) real(stnd) On entry, the XYN argument of COMP_COR for the first subsample. On exit, the XYN argument of the combined sample.

XSTAT2 (INPUT) real(stnd), dimension(2) On entry, the XSTAT argument of COMP_COR for the second subsample.

YSTAT2 (INPUT) real(stnd), dimension(2) On entry, the YSTAT argument of COMP_COR for the second subsample.

XYCOR2 (INPUT) real(stnd) On entry, the XYCOR argument of COMP_COR for the second subsample.

XYN2 (INPUT) real(stnd) On entry, the XYN argument of COMP_COR for the second subsample.

Further Details

One possible application of this subroutine is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares and cross-products computed for each subsample independently using COMP_COR. The means and corrected sums of squares and cross-products for the original sample can then be calculated using UPDATE_COR. The means, variances and correlation coefficient for the original sample can be computed by a final call to COMP_COR with LAST=true.

This subroutine is adapted from:

(1) **Chan, T.F., Golub, G.H, and Leveque, R.J., 1979:** Updating formulae and a pairwise algorithm for computing sample variances. STAN-CS-79-773.

6.12.13 subroutine update_cor (xstat, ystat, xycor, xyn, xstat2, ystat2, xycor2, xyn2)

Purpose

UPDATE_COR computes sample means and corrected sums of squares and cross-products for a sample of size XYN+XYN2 given the means and corrected sums of squares and cross-products for two subsamples of size XYN and XYN2 as output by a call to COMP_COR when LAST=false on the two subsamples separetely.

The sample means, variances and coefficient correlations for the sample of size XYN+XYN2 may be obtained by a call to COMP_COR with LAST=true.

Arguments

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,2) On entry, the XSTAT argument of COMP_COR for the first subsample. On exit, the XSTAT argument of the combined sample.

YSTAT (INPUT/OUTPUT) real(stnd), dimension(2) On entry, the YSTAT argument of COMP_COR for the first subsample. On exit, the YSTAT argument of the combined sample.

XYCOR (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the XYCOR argument of COMP_COR for the first subsample. On exit, the XYCOR argument of the combined sample.

XYN (INPUT/OUTPUT) real(stnd) On entry, the XYN argument of COMP_COR for the first subsample. On exit, the XYN argument of the combined sample.

XSTAT2 (INPUT) real(stnd), dimension(:,2) On entry, the XSTAT argument of COMP_COR for the second subsample.

YSTAT2 (INPUT) real(stnd), dimension(2) On entry, the YSTAT argument of COMP_COR for the second subsample.

XYCOR2 (INPUT) real(stnd), dimension(:) On entry, the XYCOR argument of COMP_COR for the second subsample.

XYN2 (INPUT) real(stnd) On entry, the XYN argument of COMP_COR for the second subsample.

Further Details

One possible application of this subroutine is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares and cross-products computed for each subsample independently using COMP_COR. The means and corrected sums of squares and cross-products for the original sample can then be calculated using UPDATE_COR. The means, variances and correlation coefficient for the original sample can be computed by a final call to COMP_COR with LAST=true.

This subroutine is adapted from:

(1) **Chan, T.F., Golub, G.H, and Leveque, R.J., 1979:** Updating formulae and a pairwise algorithm for computing sample variances. STAN-CS-79-773.

6.12.14 subroutine update_cor (xstat, ystat, xycor, xyn, xstat2, ystat2, xycor2, xyn2)

Purpose

UPDATE_COR computes sample means and corrected sums of squares and cross-products for a sample of size XYN+XYN2 given the means and corrected sums of squares and cross-products for two subsamples of size XYN and XYN2 as output by a call to COMP_COR when LAST=false on the two subsamples separetely.

The sample means, variances and coefficient correlations for the sample of size XYN+XYN2 may be obtained by a call to COMP_COR with LAST=true.

Arguments

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,:,2) On entry, the XSTAT argument of COMP_COR for the first subsample. On exit, the XSTAT argument of the combined sample.

YSTAT (INPUT/OUTPUT) real(stnd), dimension(2) On entry, the YSTAT argument of COMP_COR for the first subsample. On exit, the YSTAT argument of the combined sample.

XYCOR (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the XYCOR argument of COMP_COR for the first subsample. On exit, the XYCOR argument of the combined sample.

XYN (INPUT/OUTPUT) real(stnd) On entry, the XYN argument of COMP_COR for the first subsample. On exit, the XYN argument of the combined sample.

XSTAT2 (INPUT) real(stnd), dimension(:,:,2) On entry, the XSTAT argument of COMP_COR for the second subsample.

YSTAT2 (INPUT) real(stnd), dimension(2) On entry, the YSTAT argument of COMP_COR for the second subsample.

XYCOR2 (INPUT) real(stnd), dimension(:,:) On entry, the XYCOR argument of COMP_COR for the second subsample.

XYN2 (INPUT) real(stnd) On entry, the XYN argument of COMP_COR for the second subsample.

Further Details

One possible application of this subroutine is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares and cross-products computed for each subsample independently using COMP_COR. The means and corrected sums of squares and cross-products for the original sample can then be calculated using UPDATE_COR. The means, variances and correlation coefficient for the original sample can be computed by a final call to COMP_COR with LAST=true.

This subroutine is adapted from:

(1) **Chan, T.F., Golub, G.H, and Leveque, R.J., 1979:** Updating formulae and a pairwise algorithm for computing sample variances. STAN-CS-79-773.

6.12.15 subroutine update_cor_miss2 (xstat, ystat, xycor, xyn, xstat2, ystat2, xycor2, xyn2)

Purpose

UPDATE_COR_MISS2 computes sample means and corrected sums of squares and cross-products for a sample of size XYN+XYN2, possibly containing missing values, given the means and corrected sums of squares and cross-products for two subsamples of size XYN and XYN2 as output by a call to COMP_COR_MISS2 when LAST=false on the two subsamples separetely.

The sample means, variances and coefficient correlations for the sample of size XYN+XYN2 may be obtained by a call to COMP_COR_MISS2 with LAST=true.

Arguments

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,2) On entry, the XSTAT argument of COMP_COR_MISS2 for the first subsample. On exit, the XSTAT argument of the combined sample.

YSTAT (INPUT/OUTPUT) real(stnd), dimension(:,2) On entry, the YSTAT argument of COMP_COR_MISS2 for the first subsample. On exit, the YSTAT argument of the combined sample.

XYCOR (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the XYCOR argument of COMP_COR_MISS2 for the first subsample. On exit, the XYCOR argument of the combined sample.

XYN (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the XYN argument of COMP_COR_MISS2 for the first subsample. On exit, the XYN argument of the combined sample.

XSTAT2 (INPUT) real(stnd), dimension(:,2) On entry, the XSTAT argument of COMP_COR_MISS2 for the second subsample.

YSTAT2 (INPUT) real(stnd), dimension(:,2) On entry, the YSTAT argument of COMP_COR_MISS2 for the second subsample.

XYCOR2 (INPUT) real(stnd), dimension(:) On entry, the XYCOR argument of COMP_COR_MISS2 for the second subsample.

XYN2 (INPUT) real(stnd), dimension(:) On entry, the XYN argument of COMP_COR_MISS2 for the second subsample.

Further Details

One possible application of this subroutine is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares and cross-products computed for each subsample independently using COMP_COR_MISS2. The means and corrected sums of squares and cross-products for the original sample can then be calculated using UPDATE_COR_MISS2. The means, variances and correlation coefficient for the original sample can be computed by a final call to COMP_COR_MISS2 with LAST=true.

This subroutine is adapted from:

(1) **Chan, T.F., Golub, G.H, and Leveque, R.J., 1979:** Updating formulae and a pairwise algorithm for computing sample variances. STAN-CS-79-773.

6.12.16 subroutine update_cor_miss2 (xstat, ystat, xycor, xyn, xstat2, ystat2, xycor2, xyn2)

Purpose

UPDATE_COR_MISS2 computes sample means and corrected sums of squares and cross-products for a sample of size XYN+XYN2, possibly containing missing values, given the means and corrected sums of squares and cross-products for two subsamples of size XYN and XYN2 as output by a call to COMP_COR_MISS2 when LAST=false on the two subsamples separetely.

The sample means, variances and coefficient correlations for the sample of size XYN+XYN2 may be obtained by a call to COMP_COR_MISS2 with LAST=true.

Arguments

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,:,2) On entry, the XSTAT argument of COMP_COR_MISS2 for the first subsample. On exit, the XSTAT argument of the combined sample.

YSTAT (INPUT/OUTPUT) real(stnd), dimension(:,:,2) On entry, the YSTAT argument of COMP_COR_MISS2 for the first subsample. On exit, the YSTAT argument of the combined sample.

XYCOR (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the XYCOR argument of COMP_COR_MISS2 for the first subsample. On exit, the XYCOR argument of the combined sample.

XYN (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the XYN argument of COMP_COR_MISS2 for the first subsample. On exit, the XYN argument of the combined sample.

XSTAT2 (INPUT) real(stnd), dimension(:,:,2) On entry, the XSTAT argument of COMP_COR_MISS2 for the second subsample.

YSTAT2 (INPUT) real(stnd), dimension(:,:,2) On entry, the YSTAT argument of COMP_COR_MISS2 for the second subsample.

XYCOR2 (INPUT) real(stnd), dimension(:,:) On entry, the XYCOR argument of COMP_COR_MISS2 for the second subsample.

XYN2 (INPUT) real(stnd), dimension(:,:) On entry, the XYN argument of COMP_COR_MISS2 for the second subsample.

Further Details

One possible application of this subroutine is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares and cross-products computed for each subsample independently using COMP_COR_MISS2. The means and corrected sums of squares and cross-products for the original sample can then be calculated using UPDATE_COR_MISS2. The means, variances and correlation coefficient for the original sample can be computed by a final call to COMP_COR_MISS2 with LAST=true.

This subroutine is adapted from:

(1) **Chan, T.F., Golub, G.H, and Leveque, R.J., 1979:** Updating formulae and a pairwise algorithm for computing sample variances. STAN-CS-79-773.

6.12.17 `subroutine permute_cor ( x, y, xstat, ystat, xycor, prob, nrep, initseed )`

Purpose

PERMUTE_COR performs a permutation test of a correlation coefficient between two data vectors Y and X.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the input data vector X. On exit, the data are standardized with the univariate statistics stored in the XSTAT argument.

Y (INPUT) real(stnd), dimension(:) On entry, the input data vector Y.

The size of Y must verify: SIZE(Y) = SIZE(X) .

XSTAT (INPUT) real(stnd), dimension(2) On entry, XSTAT must contain the following statistics as output by COMP_COR subroutine in argument XSTAT:

- XSTAT(1) contains the mean value of the data vector X.
- XSTAT(2) contains the variance of the data vector X.

The size of XSTAT must verify: size(XSTAT) = 2.

YSTAT (INPUT) real(stnd), dimension(2) On entry, YSTAT must contain the following statistics, as output by COMP_COR subroutine in argument YSTAT:

- YSTAT(1) contains the mean value of the data vector Y.
- YSTAT(2) contains the variance of the data vector Y.

The size of YSTAT must verify: size(YSTAT) = 2.

XYCOR (INPUT) real(stnd) On entry, XYCOR contains the correlation coefficient between X(:) and Y(:). XYCOR must be specified as output by COMP_COR subroutine.

PROB (OUTPUT) real(stnd) On exit, PROB gives the critical probability associated with XYCOR, as computed by a permutation test with NREP shuffles.

NREP (INPUT, OPTIONAL) integer(i4b) On entry, when argument NREP is present, NREP specifies the number of shuffles for the permutation test of the correlation coefficient.

The default is 99.

INITSEED (INPUT, OPTIONAL) logical(lgl) On entry, if INITSEED=true, a call to RANDOM_SEED_() without arguments is done in the subroutine, in order to initiates a non-repeatable reset of the seed used by the STATPACK random generator.

The default is INITSEED=false.

Further Details

This subroutine is parallelized if OPENMP is used.

For more details and algorithm, see:

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapters 5 and 8, 484 pp., ISBN:9780521012300

(2) **Noreen, E.W., 1989:** Computer-intensive methods for testing hypotheses: an introduction. Wiley and Sons, New York, USA, ISBN:978-0-471-61136-3

6.12.18 subroutine permute_cor (x, y, xstat, ystat, xycor, prob, dimvar, nrep, initseed)

Purpose

PERMUTE_COR performs permutation tests of correlation coefficients between a data vector Y and a data matrix X.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, input matrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables for which permutation tests are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details.

On exit, the data are standardized with the univariate statistics stored in the XSTAT argument.

Y (INPUT) real(stnd), dimension(:) On entry, input vector containing size(X,3-DIMVAR) observations for which permutation tests are desired.

The size of Y must verify: size(Y) = SIZE(X, 3-DIMVAR) .

XSTAT (INPUT) real(stnd), dimension(:,2) On entry, XSTAT must contain the following statistics on all variables, as output by COMP_COR subroutine in argument XSTAT:

- XSTAT(:,1) contains the mean values of the data matrix X.
- XSTAT(:,2) contains the variances of the data matrix X.

The shape of XSTAT must verify:

- size(XSTAT, 1) = size(X, DIMVAR) ,
- size(XSTAT, 2) = 2 .

YSTAT (INPUT) real(stnd), dimension(2) On entry, YSTAT must contain the following statistics, as output by COMP_COR subroutine in argument YSTAT:

- YSTAT(1) contains the mean value of the data vector Y.
- YSTAT(2) contains the variance of the data vector Y.

The size of YSTAT must verify: size(YSTAT) = 2.

XYCOR (INPUT) real(stnd), dimension(:) On entry, XYCOR(i) contains the correlation coefficient between XX(i,:) (XX(:,i) if DIMVAR=2) and YY(:). XYCOR must be specified as output by COMP_COR subroutine.

The size of XYCOR must verify: size(XYCOR) = size(X, DIMVAR).

PROB (OUTPUT) real(stnd), dimension(:) On exit, PROB(i) gives the critical probability associated with XYCOR(i), as computed by a permutation test with NREP shuffles.

The size of PROB must verify: size(PROB) = size(X, DIMVAR).

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows:

- DIMVAR = 1, the input matrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input matrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

NREP (INPUT, OPTIONAL) integer(i4b) On entry, when argument NREP is present, NREP specifies the number of shuffles for the permutation test of the correlation coefficients.

The default is 99.

INITSEED (INPUT, OPTIONAL) logical(lgl) On entry, if INITSEED=true, a call to RANDOM_SEED_() without arguments is done in the subroutine, in order to initiates a non-repeatable reset of the seed used by the STATPACK random generator.

The default is INITSEED=false.

Further Details

This subroutine is parallelized if OPENMP is used.

For more details and algorithm, see:

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapters 5 and 8, 484 pp., ISBN:9780521012300

(2) **Noreen, E.W., 1989:** Computer-intensive methods for testing hypotheses: an introduction. Wiley and Sons, New York, USA, ISBN:978-0-471-61136-3

6.12.19 subroutine phase_scramble_cor (x, y, xstat, ystat, xycor, prob, nrep, method, norm, initseed)

Purpose

PHASE_SCRAMBLE_COR performs phase-scrambled bootstrap tests of a correlation coefficient between two data vectors Y and X.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the input data vector X. On exit, the data are standardized with the univariate statistics stored in the XSTAT argument.

Y (INPUT) real(stnd), dimension(:) On entry, the input data vector Y.

The size of Y must verify: SIZE(Y) = SIZE(X) .

XSTAT (INPUT) real(stnd), dimension(2) On entry, XSTAT must contain the following statistics as output by COMP_COR subroutine in argument XSTAT:

- XSTAT(1) contains the mean value of the data vector X.
- XSTAT(2) contains the variance of the data vector X.

The size of XSTAT must verify: size(XSTAT) = 2.

YSTAT (INPUT) real(stnd), dimension(2) On entry, YSTAT must contain the following statistics, as output by COMP_COR subroutine in argument YSTAT:

- YSTAT(1) contains the mean value of the data vector Y.
- YSTAT(2) contains the variance of the data vector Y.

The size of YSTAT must verify: size(YSTAT) = 2.

XYCOR (INPUT) real(stnd) On entry, XYCOR contains the correlation coefficient between X(:) and Y(:). XYCOR must be specified as output by COMP_COR subroutine.

PROB (OUTPUT) real(stnd) On exit, PROB gives the critical probability associated with XYCOR, as computed by a phase-scrambled bootstrap test with NREP shuffles.

NREP (INPUT, OPTIONAL) integer(i4b) On entry, when argument NREP is present, NREP specifies the number of shuffles for the phase-scrambled bootstrap test of the correlation coefficient.

The default is 99.

METHOD (INPUT, OPTIONAL) integer(i4b) On entry, determine the phase randomisation algorithm used to generate surrogate series.

On entry, if

- METHOD = 1 : the phase randomisation algorithm of Theiler is used;
- METHOD = 2 : the phase randomisation algorithm of Davison and Hinkley is used.

The default is METHOD = 1.

NORM (INPUT, OPTIONAL) logical(lgl) On entry, if NORM=true, then normal margins are used in the phase-scrambled algorithm, otherwise exact empirical margins are used.

The default is NORM=true.

INITSEED (INPUT, OPTIONAL) logical(lgl) On entry, if INITSEED=true, a call to RANDOM_SEED_() without arguments is done in the subroutine, in order to initiates a non-repeatable reset of the seed used by the STATPACK random generator.

The default is INITSEED=false.

Further Details

This subroutine is parallelized if OPENMP is used.

The tests are adapted from:

(1) **Ebisuzaki, W., 1997:** A method to estimate the statistical significance of a correlation when the data are serially correlated. Journal of climate, vol. 10, 2147-2153.

(2) **Davison, A.C., and Hinkley, D.V., 1997:** Bootstrap methods and their application. Cambridge University Press, Cambridge, UK. doi:10.1017/CBO9780511802843

(3) **Theiler, J., Eubank, S., Longtin, A., Galdrikian, B., and Farmer, J.D., 1992:** Testing for nonlinearity in time series: the method of surrogate data. Physica D, vol. 58, 77-94, doi:10.1016/0167-2789(92)90102-s

(4) **Braun, W.J., and Kulperger, R.J., 1997:** Properties of a fourier bootstrap method for time series. Communications in Statistics - Theory and Methods, vol 26, 1329-1336, doi:10.1080/03610929708831985

6.12.20 subroutine phase_scramble_cor (x, y, xstat, ystat, xycor, prob, dimvar, nrep, method, norm, initseed)

Purpose

PHASE_SCRAMBLE_COR performs phase-scrambled bootstrap tests of correlation coefficients between a data vector Y and a data matrix X.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, input matrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables for which phase-scrambled bootstrap tests are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details.

On exit, the data are standardized with the univariate statistics stored in the XSTAT argument.

Y (INPUT) real(stnd), dimension(:) On entry, input vector containing size(X,3-DIMVAR) observations for which phase-scrambled bootstrap tests are desired.

The size of Y must verify: size(Y) = SIZE(X, 3-DIMVAR) .

XSTAT (INPUT) real(stnd), dimension(:,2) On entry, XSTAT must contain the following statistics on all variables, as output by COMP_COR subroutine in argument XSTAT:

- XSTAT(:,1) contains the mean values of the data matrix X.
- XSTAT(:,2) contains the variances of the data matrix X.

The shape of XSTAT must verify:

- size(XSTAT, 1) = size(X, DIMVAR) ,
- size(XSTAT, 2) = 2 .

YSTAT (INPUT) real(stnd), dimension(2) On entry, YSTAT must contain the following statistics, as output by COMP_COR subroutine in argument YSTAT:

- YSTAT(1) contains the mean value of the data vector Y.
- YSTAT(2) contains the variance of the data vector Y.

The size of YSTAT must verify: size(YSTAT) = 2.

XYCOR (INPUT) real(stnd), dimension(:) On entry, XYCOR(i) contains the correlation coefficient between XX(i,:) (XX(:,i) if DIMVAR=2) and YY(:). XYCOR must be specified as output by COMP_COR subroutine.

The size of XYCOR must verify: size(XYCOR) = size(X, DIMVAR).

PROB (OUTPUT) real(stnd), dimension(:) On exit, PROB(i) gives the critical probability associated with XYCOR(i), as computed by a phase-scrambled bootstrap test with NREP shuffles.

The size of PROB must verify: size(PROB) = size(X, DIMVAR).

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows, if:

- DIMVAR = 1, the input matrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input matrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

NREP (INPUT, OPTIONAL) integer(i4b) On entry, when argument NREP is present, NREP specifies the number of shuffles for the phase-scrambled bootstrap test of the correlation coefficients.

The default is 99.

METHOD (INPUT, OPTIONAL) integer(i4b) On entry, determine the phase randomisation algorithm used to generate surrogate series.

On entry, if

- METHOD = 1 : the phase randomisation algorithm of Theiler is used;
- METHOD = 2 : the phase randomisation algorithm of Davison and Hinkley is used.

The default is METHOD = 1.

NORM (INPUT, OPTIONAL) logical(lgl) On entry, if NORM=true, then normal margins are used in the phase-scrambled algorithm, otherwise exact empirical margins are used.

The default is NORM=true.

INITSEED (INPUT, OPTIONAL) logical(lgl) On entry, if INITSEED=true, a call to RANDOM_SEED_() without arguments is done in the subroutine, in order to initiates a non-repeatable reset of the seed used by the STATPACK random generator.

The default is INITSEED=false.

Further Details

This subroutine is parallelized if OPENMP is used.

The tests are adapted from:

(1) **Ebisuzaki, W., 1997:** A method to estimate the statistical significance of a correlation when the data are serially correlated. Journal of climate, vol. 10, 2147-2153.

(2) **Davison, A.C., and Hinkley, D.V., 1997:** Bootstrap methods and their application. Cambridge University Press, Cambridge, UK. doi:10.1017/CBO9780511802843

(3) **Theiler, J., Eubank, S., Longtin, A., Galdrikian, B., and Farmer, J.D., 1992:** Testing for nonlinearity in time series: the method of surrogate data. Physica D, vol. 58, 77-94, doi:10.1016/0167-2789(92)90102-s

(4) **Braun, W.J., and Kulperger, R.J., 1997:** Properties of a fourier bootstrap method for time series. Communications in Statistics - Theory and Methods, vol 26, 1329-1336, doi:10.1080/03610929708831985

6.12.21 subroutine bootstrap_cor (x, y, xstat, xycor, prob, nrep, initseed, periodicity, season, block_size)

Purpose

BOOTSTRAP_COR performs a moving block bootstrap test of a correlation coefficient between two data vectors X and Y.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the input data vector X.

On exit, the data are standardized with the univariate statistics stored in the XSTAT argument.

Y (INPUT) real(stnd), dimension(:) On entry, the input data vector Y.

The size of Y must verify: SIZE(Y) = SIZE(X) .

XSTAT (INPUT) real(stnd), dimension(2) On entry, XSTAT must contain the following statistics as output by COMP_COR subroutine in argument XSTAT:

- XSTAT(1) contains the mean value of the data vector X.
- XSTAT(2) contains the variance of the data vector X.

The size of XSTAT must verify: size(XSTAT) = 2.

XYCOR (INPUT) real(stnd) On entry, XYCOR contains the correlation coefficient between X(:) and Y(:). XYCOR must be specified as output by COMP_COR subroutine.

PROB (OUTPUT) real(stnd) On exit, PROB gives the critical probability associated with XYCOR, as computed by a moving block bootstrap test with NREP shuffles.

NREP (INPUT, OPTIONAL) integer(i4b) On entry, when argument NREP is present, NREP specifies the number of shuffles for the moving block bootstrap test of the correlation coefficient.

The default is 99.

INITSEED (INPUT, OPTIONAL) logical(lgl) On entry, if INITSEED=true, a call to RANDOM_SEED_() without arguments is done in the subroutine, in order to initiates a non-repeatable reset of the seed used by the STATPACK random generator.

The default is INITSEED=false.

PERIODICITY (INPUT, OPTIONAL) integer(i4b) On entry, argument PERIODICITY specifies that the indice, i, of the first observation of each selected block in the moving block bootstrap algorithm verifies the condition i=1+(PERIODICITY * j) where j is a random positive integer. PERIODICITY must be greater than zero and less than size(X).

By default, PERIODICITY is set to 1.

SEASON (INPUT, OPTIONAL) integer(i4b) On entry, argument SEASON specifies that the input time series is a repetition of the same season for different years and SEASON specifies the length of the season. SEASON must be greater than zero and size(X) must be a multiple of SEASON. If the optional argument PERIODICITY is used, SEASON must also be greater or equal to PERIODICITY.

By default, SEASON is set to size(X).

BLOCK_SIZE (INPUT, OPTIONAL) integer(i4b) On entry, argument BLOCK_SIZE specifies the size of the block in the moving block bootstrap. BLOCK_SIZE must be greater than zero and less than size(X). If the optional argument PERIODICITY is used, BLOCK_SIZE must also be greater or equal to PERIODICITY. Moreover, if the optional argument SEASON is used, BLOCK_SIZE must also be less than SEASON.

By default, BLOCK_SIZE is set to 1 or to PERIODICITY if this optional argument is used.

Further Details

This subroutine is parallelized if OPENMP is used.

The test is adapted from:

(1) **Davison, A.C., and Hinkley, D.V., 1997:** Bootstrap methods and their application. Cambridge University Press, Cambridge, UK. doi:10.1017/CBO9780511802843

6.12.22 subroutine bootstrap_cor (x, y, xstat, xycor, prob, dimvar, nrep, initseed, periodicity, season, block_size)

Purpose

BOOTSTRAP_COR performs a moving block bootstrap test of a correlation coefficients between a data vector Y and a data matrix X.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, input matrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables for which moving block bootstrap tests are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details.

On exit, the data are standardized with the univariate statistics stored in the XSTAT argument.

Y (INPUT) real(stnd), dimension(:) On entry, input vector containing size(X,3-DIMVAR) observations for which moving block bootstrap tests are desired.

The size of Y must verify: size(Y) = size(X, 3-DIMVAR) .

XSTAT (INPUT) real(stnd), dimension(:,2) On entry, XSTAT must contain the following statistics on all variables, as output by COMP_COR subroutine in argument XSTAT:

- XSTAT(:,1) contains the mean values of the data matrix X.
- XSTAT(:,2) contains the variances of the data matrix X.

The shape of XSTAT must verify:

- size(XSTAT, 1) = size(X, DIMVAR) ,
- size(XSTAT, 2) = 2 .

XYCOR (INPUT) real(stnd), dimension(:) On entry, XYCOR(i) contains the correlation coefficient between XX(i,:) (XX(:,i) if DIMVAR=2) and YY(:). XYCOR must be specified as output by COMP_COR subroutine.

The size of XYCOR must verify: size(XYCOR) = size(X, DIMVAR).

PROB (OUTPUT) real(stnd), dimension(:) On exit, PROB(i) gives the critical probability associated with XYCOR(i), as computed by a moving block bootstrap test with NREP shuffles.

The size of PROB must verify: size(PROB) = size(X, DIMVAR).

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows, if:

- DIMVAR = 1, the input matrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input matrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

NREP (INPUT, OPTIONAL) integer(i4b) On entry, when argument NREP is present, NREP specifies the number of shuffles for the moving block bootstrap test of the correlation coefficient.

The default is 99.

INITSEED (INPUT, OPTIONAL) logical(lgl) On entry, if INITSEED=true, a call to RANDOM_SEED_() without arguments is done in the subroutine, in order to initiates a non-repeatable reset of the seed used by the STATPACK random generator.

The default is INITSEED=false.

PERIODICITY (INPUT, OPTIONAL) integer(i4b) On entry, argument PERIODICITY specifies that the indice, i, of the first observation of each selected block in the moving block bootstrap algorithm verifies the condition i=1+(PERIODICITY * j) where j is a random positive integer. PERIODICITY must be greater than zero and less than size(X,3-DIMVAR).

By default, PERIODICITY is set to 1.

SEASON (INPUT, OPTIONAL) integer(i4b) On entry, argument SEASON specifies that the input time series is a repetition of the same season for different years and SEASON specifies the length of the season. SEASON must be greater than zero and size(X) must be a multiple of SEASON. If the optional argument PERIODICITY is used, SEASON must also be greater or equal to PERIODICITY.

By default, SEASON is set to size(X,3-DIMVAR).

BLOCK_SIZE (INPUT, OPTIONAL) integer(i4b) On entry, argument BLOCK_SIZE specifies the size of the block in the moving block bootstrap. BLOCK_SIZE must be greater than zero and less than size(X). If the optional argument PERIODICITY is used, BLOCK_SIZE must also be greater or equal to PERIODICITY. Moreover, if the optional argument SEASON is used, BLOCK_SIZE must also be less than SEASON.

By default, BLOCK_SIZE is set to 1 or to PERIODICITY if this optional argument is used.

Further Details

This subroutine is parallelized if OPENMP is used.

The test is adapted from:

(1) **Davison, A.C., and Hinkley, D.V., 1997:** Bootstrap methods and their application. Cambridge University Press, Cambridge, UK. doi:10.1017/CBO9780511802843

6.12.23 subroutine comp_cormat (x, first, last, xmean, xcor, xn, dimvar, xstd, cov, fill, failure)

Purpose

COMP_CORMAT computes estimates of means and variance-covariance or correlation matrix from a data matrix.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which means, variances and co-

variances are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrix is the first submatrix of the data matrix.
- FIRST = false the current submatrix is not the first submatrix of the data matrix.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrix is the last submatrix of the data matrix.
- LAST = false the current submatrix is not the last submatrix of the data matrix.

XMEAN (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_CORMAT (e.g., when when FIRST=true), XMEAN contains the variable means from previous calls to COMP_CORMAT. XMEAN should not be changed between calls to COMP_CORMAT.

On exit, when LAST=true, XMEAN contains the variable means

The size of XMEAN must verify: size(XMEAN) = size(X, DIMVAR).

XCOR (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_CORMAT (e.g., when when FIRST=true), the matrix XCOR contains the upper triangle of the corrected sums of squared and cross-products matrix computed from previous calls to COMP_CORMAT. XCOR should not be changed between calls to COMP_CORMAT.

On exit, when LAST=true, XCOR contains the upper triangle of the symetric correlation or variance-covariance matrix as controlled by the COV argument. If the optional argument FILL is present and equal to true, the lower triangle of XCOR is also filled.

The shape of XCOR must verify:

- size(XCOR, 1) = size(X, DIMVAR) ,
- size(XCOR, 2) = size(X, DIMVAR) .

XN (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_CORMAT (e.g., when when FIRST=true), XN contains count of observations from previous calls to COMP_CORMAT. XN should not be changed between calls to COMP_CORMAT.

On exit, XN contains the number of observations in the data matrix XX.

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows, if:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

XSTD (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XSTD is present, XSTD contains the variable standard-deviations.

The size of XSTD must verify: size(XSTD) = size(X, DIMVAR).

XSTD needs to be specified only on the last call to COMP_CORMAT (e.g., when LAST=true).

COV (INPUT, OPTIONAL) logical(lgl) On entry, when argument COV is present, COV is used as follows, if:

- COV= true, XCOR contains the variances-covariances matrix, when LAST=true.
- COV= false, XCOR contains the correlation matrix, when LAST=true.

By default, the correlation matrix is output.

COV needs to be specified only on the last call to COMP_CORMAT (e.g., when LAST=true).

FILL (INPUT, OPTIONAL) logical(lgl) On entry, when argument FILL is present, FILL is used as follows, if:

- FILL= true, the lower triangle of XCOR is filled, when LAST=true.
- FILL= false, the lower triangle of XCOR is not filled, when LAST=true.

By default, the lower triangle of XCOR is not filled.

FILL needs to be specified only on the last call to COMP_CORMAT (e.g., when LAST=true).

FAILURE (OUTPUT, OPTIONAL) logical(lgl) On exit, when argument FAILURE is present:

- FAILURE = false: indicates successful exit.
- FAILURE = true: indicates that fewer than two valid observations were present or that the observations on some variable were constant and the correlations were requested.

Further Details

The subroutine computes the means and correlation matrix with only one pass through the data.

If the observations on some variable were constant, the pertinent correlations are set to nan() code .

If fewer than two valid observations were present, the correlations are set to nan() code .

If fewer than one valid observation is present, the means are also set to nan() code .

For more details on correlation analysis, see

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 8, 484 pp., ISBN:9780521012300

6.12.24 subroutine comp_cormat (x, first, last, xmean, xcorp, xn, dimvar, xstd, cov, failure)

Purpose

COMP_CORMAT computes estimates of means and variance-covariance or correlation matrix from a data matrix.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which means, variances and covariances are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrix is the first submatrix of the data matrix.
- FIRST = false the current submatrix is not the first submatrix of the data matrix.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrix is the last submatrix of the data matrix.

- LAST = false the current submatrix is not the last submatrix of the data matrix.

XMEAN (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_CORMAT (e.g., when when FIRST=true), XMEAN contains the variable means from previous calls to COMP_CORMAT. XMEAN should not be changed between calls to COMP_CORMAT. On exit, when LAST=true, XMEAN contains the variable means

The size of XMEAN must verify: size(XMEAN) = size(X, DIMVAR).

XCORP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_CORMAT (e.g., when when FIRST=true), the linear array XCORP contains the upper triangle of the corrected sums of squared and cross-products matrix, packed columnwise, computed from previous calls to COMP_CORMAT. XCORP should not be changed between calls to COMP_CORMAT.

On exit, when LAST=true, XCORP contains the correlation or variance-covariance matrix as controlled by the COV argument. XCORP is stored in symmetric storage mode (see further details).

The size of XCORP must verify: size(XCORP) = (size(X,DIMVAR) * (size(X,DIMVAR)+1))/2.

XN (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_CORMAT (e.g., when when FIRST=true), XN contains count of observations from previous calls to COMP_CORMAT. XN should not be changed between calls to COMP_CORMAT.

On exit, XN contains the number of observations in the data matrix XX.

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows, if:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

XSTD (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XSTD is present, XSTD contains the variable standard-deviations.

The size of XSTD must verify: size(XSTD) = size(X, DIMVAR).

XSTD needs to be specified only on the last call to COMP_CORMAT (e.g., when LAST=true).

COV (INPUT, OPTIONAL) logical(lgl) On entry, when argument COV is present, COV is used as follows, if:

- COV= true, XCORP contains the variances-covariances matrix, when LAST=true.
- COV= false, XCORP contains the correlation matrix, when LAST=true.

By default, the correlation matrix is output.

COV needs to be specified only on the last call to COMP_CORMAT (e.g., when LAST=true).

FAILURE (OUTPUT, OPTIONAL) logical(lgl) On exit, when argument FAILURE is present:

- FAILURE = false: indicates successful exit.
- FAILURE = true: indicates that fewer than two valid observations were present or that the observations on some variable were constant and the correlations were requested.

Further Details

The subroutine computes the means and the correlation matrix with only one pass through the data.

On exit, the upper triangle of the symmetric correlation or variance-covariance matrix XCOR is packed columnwise in the linear array XCORP. More precisely, the j-th column of XCOR is stored in the array XCORP as follows:

XCORP(i + (j-1) * j/2) = XCOR(i,j) for 1<=i<=j;

If the observations on some variable were constant, the pertinent correlations are set to nan() code .

If fewer than two valid observations were present, the correlations are set to nan() code .

If fewer than one valid observation is present, the means are also set to nan() code .

For more details on correlation analysis, see:

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 8, 484 pp., ISBN:9780521012300

6.12.25 subroutine comp_cormat_miss (x, first, last, xmean, xcor, xn, xmiss, dimvar, xstd, cov, fill, failure)

Purpose

COMP_CORMAT_MISS computes estimates of means and variance-covariance or correlation matrix from a data matrix possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which means, variances and covariances are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrix is the first submatrix of the data matrix.
- FIRST = false the current submatrix is not the first submatrix of the data matrix.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrix is the last submatrix of the data matrix.
- LAST = false the current submatrix is not the last submatrix of the data matrix.

XMEAN (INPUT/OUTPUT) real(stnd), dimension(:,2) On entry, after the first call to COMP_CORMAT_MISS (e.g., when FIRST=true), XMEAN(:,1) contains the variable means from previous calls to COMP_CORMAT_MISS. XMEAN should not be changed between calls to COMP_CORMAT_MISS.

On exit, when LAST=true, XMEAN(:,1) contains the variable means computed from all non-missing observations in the data matrix. XMEAN(:,2) is used as workspace.

The shape of XMEAN must verify:

- size(XMEAN, 1) = size(X, DIMVAR) ,
- size(MEAN, 2) = 2 .

XCOR (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_CORMAT_MISS (e.g., when FIRST=true), the matrix XCOR contains the upper

triangle of the corrected sums of squared and cross-products matrix computed from previous calls to COMP_CORMAT_MISS. XCOR should not be changed between calls to COMP_CORMAT_MISS.

On exit, when LAST=true, XCOR contains the upper triangle of the symetric correlation or variance-covariance matrix as controlled by the COV argument. If the optional argument FILL is present and equal to true, the lower triangle of XCOR is also filled.

The shape of XCOR must verify:

- size(XCOR, 1) = size(X, DIMVAR) ,
- size(XCOR, 2) = size(X, DIMVAR) .

XN (INPUT/OUTPUT) real(stnd), dimension(:,3) On entry, after the first call to COMP_CORMAT_MISS (e.g., when FIRST=true), XN is used as workspace to accumulate quantities from previous calls to COMP_CORMAT_MISS. XN should not be changed between calls to COMP_CORMAT_MISS.

On exit, XN(:,1) contains the upper triangle of the matrix of the incidence values between each pair of variables, packed columnwise, in a linear array. XN(i + (j-1) * j/2,1) indicates the numbers of non-missing pairs which were used in the calculation of XCOR(i,j) for 1<=i<=j . XN(:,2:3) is used as workspace.

The shape of XN must verify:

- size(XN, 1) = (size(X,DIMVAR) * (size(X,DIMVAR)+1))/2 ,
- size(XN, 2) = 3 .

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing or invalid. The means and the correlations are computed on all the observations where X are not missing (see Further Details).

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows, if:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

XSTD (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XSTD is present, XSTD contains the variable standard-deviations.

The size of XSTD must verify: size(XSTD) = size(X, DIMVAR).

XSTD needs to be specified only on the last call to COMP_CORMAT_MISS (e.g., when LAST=true).

COV (INPUT, OPTIONAL) logical(lgl) On entry, when argument COV is present, COV is used as follows, if:

- COV= true, XCORP contains the variances-covariances matrix, when LAST=true.
- COV= false, XCORP contains the correlation matrix, when LAST=true.

By default, the correlation matrix is output.

COV needs to be specified only on the last call to COMP_CORMAT_MISS (e.g., when LAST=true).

FILL (INPUT, OPTIONAL) logical(lgl) On entry, when argument FILL is present, FILL is used as follows, if:

- FILL= true, the lower triangle of XCOR is filled, when LAST=true.

- FILL= false, the lower triangle of XCOR is not filled, when LAST=true.

By default, the lower triangle of XCOR is not filled.

FILL needs to be specified only on the last call to COMP_CORMAT_MISS (e.g., when LAST=true).

FAILURE (OUTPUT, OPTIONAL) logical(lgl) On exit, when argument FAILURE is present:

- FAILURE = false: indicates successful exit.
- FAILURE = true: indicates that fewer than two valid observations were present for some pair of variables or that the observations on some variable were constant and the correlations were requested.

Further Details

The subroutine computes the means and the correlation matrix with only one pass through the data.

If the observations on some variable were constant, the pertinent correlations are set to XMISS.

If fewer than two valid observations were present for some pair of variables, the pertinent correlations are set to XMISS.

If fewer than one valid observation is present for some variables, the pertinent means are also set to XMISS.

The means and standard-deviations of the data matrix are computed from all valid data. The correlation coefficients are based on these univariate statistics and on all valid pairs of observations.

For more details on correlation analysis, see:

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 8, 484 pp., ISBN:9780521012300

6.12.26 `subroutine comp_cormat_miss ( x, first, last, xmean, xcorp, xn, xmiss, dimvar, xstd, cov, failure )`

Purpose

COMP_CORMAT_MISS computes estimates of means and variance-covariance or correlation matrix from a data matrix possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which means, variances and covariances are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrix is the first submatrix of the data matrix.
- FIRST = false the current submatrix is not the first submatrix of the data matrix.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrix is the last submatrix of the data matrix.
- LAST = false the current submatrix is not the last submatrix of the data matrix.

XMEAN (INPUT/OUTPUT) real(stnd), dimension(:,2) On entry, after the first call to COMP_CORMAT_MISS (e.g., when FIRST=true), XMEAN(:,1) contains the variable means from previous calls to COMP_CORMAT_MISS. XMEAN should not be changed between calls to COMP_CORMAT_MISS.

On exit, when LAST=true, XMEAN(:,1) contains the variable means computed from all non-missing observations in the data matrix. XMEAN(:,2) is used as workspace.

The shape of XMEAN must verify:

- size(XMEAN, 1) = size(X, DIMVAR) ,
- size(MEAN, 2) = 2 .

XCORP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_CORMAT_MISS (e.g., when FIRST=true), the linear array XCORP contains the upper triangle of the corrected sums of squared and cross-products matrix, packed columnwise, computed from previous calls to COMP_CORMAT_MISS. XCORP should not be changed between calls to COMP_CORMAT_MISS.

On exit, when LAST=true, XCORP contains the correlation or variance-covariance matrix as controlled by the COV argument. XCORP is stored in symmetric storage mode (see further details).

The size of XCORP must verify: size(XCORP) = (size(X,DIMVAR) * (size(X,DIMVAR)+1))/2.

XN (INPUT/OUTPUT) real(stnd), dimension(:,3) On entry, after the first call to COMP_CORMAT_MISS (e.g., when FIRST=true), XN is used as workspace to accumulate quantities from previous calls to COMP_CORMAT_MISS. XN should not be changed between calls to COMP_CORMAT_MISS.

On exit, XN(:,1) contains the incidence values between each pair of variables. XN(i,1) indicates the numbers of non-missing pairs of observations which were used in the calculation of XCORP(i).

The shape of XN must verify:

- size(XN, 1) = (size(X,DIMVAR) * (size(X,DIMVAR)+1))/2 ,
- size(XN, 2) = 3 .

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing or invalid. The means and the correlations are computed on all the observations where X are not missing (see Further Details).

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows, if:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

XSTD (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XSTD is present, XSTD contains the variable standard-deviations.

The size of XSTD must verify: size(XSTD) = size(X, DIMVAR).

XSTD needs to be specified only on the last call to COMP_CORMAT_MISS (e.g., when LAST=true).

COV (INPUT, OPTIONAL) logical(lgl) On entry, when argument COV is present, COV is used as follows, if:

- COV= true, XCORP contains the variances-covariances matrix, when LAST=true.

- COV= false, XCORP contains the correlation matrix, when LAST=true.

By default, the correlation matrix is output.

COV needs to be specified only on the last call to COMP_CORMAT_MISS (e.g., when LAST=true).

FAILURE (OUTPUT, OPTIONAL) logical(lgl) On exit, when argument FAILURE is present:

- FAILURE = false: indicates successful exit.
- FAILURE = true: indicates that fewer than two valid observations were present for some variable or that the observations on some variable were constant and the correlations were requested.

Further Details

The subroutine computes the means and the correlation matrix with only one pass through the data.

On exit, the upper triangle of the symmetric correlation or variance-covariance matrix XCOR is packed columnwise in the linear array XCORP. More precisely, the j-th column of XCOR is stored in the array XCORP as follows:

XCORP(i + (j-1) * j/2) = XCOR(i,j) for 1<=i<=j;

If the observations on some variable were constant, the pertinent correlations are set to XMISS .

If fewer than two valid observations were present on some variable, the pertinent correlations are set to XMISS.

If fewer than one valid observation is present for some variable, the pertinent means are also set to XMISS.

The means and standard-deviations of the data matrix are computed from all valid data. The correlation coefficients are based on these univariate statistics and on all valid pairs of observations.

For more details on correlation analysis, see:

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 8, 484 pp., ISBN:9780521012300

6.12.27 subroutine comp_eof (x, first, last, xeigval, xeigvec, xn, failure, dimvar, cov, sort, maxiter, xmean, xstd, xeigvar, xcorp)

Purpose

COMP_EOF computes estimates of Empirical Orthogonal Functions (EOF; also known as Principal Component Analysis) from a data matrix.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which Empirical Orthogonal Functions are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrix is the first submatrix of the data matrix.

- FIRST = false the current submatrix is not the first submatrix of the data matrix.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrix is the last submatrix of the data matrix.
- LAST = false the current submatrix is not the last submatrix of the data matrix.

XEIGVAL (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_EOF (e.g., when FIRST=true), XEIGVAL contains temporary results from previous calls to COMP_EOF. XEIGVAL should not be changed between calls to COMP_EOF.

On exit, when LAST=true, XEIGVAL contains the eigenvalues of th variance-covariance (or correlation) matrix from the data matrix. The near zero eigenvalues made negative by round off errors are set to zero.

The size of XEIGVAL must verify: size(XEIGVAL) = size(X, DIMVAR).

XEIGVEC (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_EOF (e.g., when FIRST=true), the matrix XEIGVEC contains temporary results from previous calls to COMP_EOF. XEIGVEC should not be changed between calls to COMP_EOF.

On exit, when LAST=true, XEIGVEC contains the eigenvectors of th variance-covariance (or correlation) matrix from the data matrix.

The shape of XEIGVEC must verify:

- size(XEIGVEC, 1) = size(X, DIMVAR) ,
- size(XEIGVEC, 2) = size(X, DIMVAR) .

XN (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_EOF (e.g., when FIRST=true), XN contains count of observations from previous calls to COMP_EOF. XN should not be changed between calls to COMP_EOF.

On exit, XN contains the number of observations in the data matrix.

FAILURE (OUTPUT) logical(lgl) On exit when LAST=true:

- FAILURE = false: indicates successful exit.
- FAILURE = true: indicates that fewer than two valid observations were present or that the observations on some variable were constant and the correlations were requested or that maximum accuracy was not achieved when computing the eigenvectors and the eigenvalues.

On exit when LAST=false, FAILURE is always set to false.

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows, if:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

COV (INPUT, OPTIONAL) logical(lgl) On entry, when argument COV is present, COV is used as follows, if:

- COV= true, XEIGVEC contains the eigenvectors of the variances-covariances matrix, when LAST=true.
- COV= false, XEIGVEC contains the eigenvectors of the correlation matrix, when LAST=true.

By default, the eigenvectors of the correlation matrix are output.

COV needs to be specified only on the last call to COMP_EOF (e.g., when LAST=true).

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The eigenvectors are reordered accordingly.

SORT needs to be specified only on the last call to COMP_EOF (e.g., when LAST=true).

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of the covariance matrix . The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(XEIGVAL). Convergence usually occurs in about 2 * size(XEIGVAL) QR sweeps.

The default is 30.

MAXITER needs to be specified only on the last call to COMP_EOF (e.g., when LAST=true).

XMEAN (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XMEAN is present, XMEAN contains the variable means.

XMEAN needs to be specified only on the last call to COMP_EOF (e.g., when LAST=true).

The size of XMEAN must verify: size(XMEAN) = size(X, DIMVAR).

XSTD (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XSTD is present, XSTD contains the variable standard-deviations.

XSTD needs to be specified only on the last call to COMP_EOF (e.g., when LAST=true).

The size of XSTD must verify: size(XSTD) = size(X, DIMVAR).

XEIGVAR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XEIGVAR is present, XEIGVAR contains percentages of total variance associated with the eigenvectors in the order of the eigenvalues stored in XEIGVAL.

XEIGVAR needs to be specified only on the last call to COMP_EOF (e.g., when LAST=true).

The size of XEIGVAR must verify: size(XEIGVAR) = size(X, DIMVAR).

XCORP (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XCORP is present, XCORP contains the upper triangle of the correlation or variance-covariance matrix, as controlled by the COV argument, stored in symmetric storage mode. The upper triangle of the symmetric correlation or variance-covariance matrix is packed columnwise in the linear array XCORP. More precisely, the j-th column of this matrix is stored in the array XCORP as follows:

XCORP(i + (j-1) * j/2,2) = XCOR(i,j) for 1<=i<=j;

XCORP needs to be specified only on the last call to COMP_EOF (e.g., when LAST=true).

The size of XCORP must verify:

- size(XCORP) = (size(X,DIMVAR) * (size(X,DIMVAR)+1))/2

Further Details

The subroutine computes the Empirical Orthogonal Functions with only one pass through the data.

This subroutine may be used in a call with no observations (e.g., size(X,3-DIMVAR) = 0) in order to finish the computations with LAST=true when the total number of observations is unknown at the beginning of the computations.

If fewer than two valid observations were present for some pair of variables or if the observations on some variable were constants, the statistics XEIGVAL, XEIGVEC, XEIGVAR and XCORP are globally set to NaN code.

For more details on EOF or PCA analysis, see:

(1) **Jackson, J.E., 2003:** A user's guide to principal components. John Wiley and Sons, New York, USA, 592 pp., ISBN:978-0-471-47134-9

(2) **Jolliffe, I.T., 2002:** Principal component analysis. Springer-Verlag, New York, USA, 487 pp., 2nd Ed, ISBN:978-0-387-22440-4

(3) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 13, 484 pp., ISBN:9780521012300

6.12.28 subroutine comp_eof2 (x, first, last, xeigval, xcorp, xn, failure, dimvar, cov, savecor, maxiter, ortho, xmean, xstd, xeigvar, xeigvec)

Purpose

COMP_EOF2 computes estimates of Empirical Orthogonal Functions (EOF; also known as Principal Component Analysis) from a data matrix.

COMP_EOF2 computes all the eigenvalues, and optionally selected eigenvectors (by inverse iteration), of the covariance (or correlation matrix) from a data matrix.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which Empirical Orthogonal Functions are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrix is the first submatrix of the data matrix.
- FIRST = false the current submatrix is not the first submatrix of the data matrix.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrix is the last submatrix of the data matrix.
- LAST = false the current submatrix is not the last submatrix of the data matrix.

XEIGVAL (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_EOF2 (e.g., when FIRST=true), XEIGVAL contains temporary results from previous calls to COMP_EOF2. XEIGVAL should not be changed between calls to COMP_EOF2.

On exit, when LAST=true, XEIGVAL contains the eigenvalues of th variance-covariance (or correlation) matrix from the data matrix. The near zero eigenvalues made negative by round off errors are set to zero. The eigenvalues are sorted in descending order. The eigenvectors in XEIGVEC are reordered accordingly.

The size of XEIGVAL must verify: size(XEIGVAL) = size(X, DIMVAR).

XCORP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_EOF2 (e.g., when FIRST=true), the linear array XCORP contains the upper triangle of the corrected sums of squared and cross-products matrix, packed columnwise, computed from previous calls to COMP_EOF2. XCORP should not be changed between calls to COMP_EOF2.

On exit, when LAST=true and SAVECOR=true, XCORP contains the correlation or variance-covariance matrix as controlled by the COV argument. XCORP is stored in symmetric storage

mode (see further details). If SAVECOR=false, the correlation matrix is not saved on exit. In this case XCORP does not contain useful information.

The size of XCORP must verify: size(XCORP) = (size(X,DIMVAR) * (size(X,DIMVAR)+1))/2.

XN (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_EOF2 (e.g., when FIRST=true), XN contains count of observations from previous calls to COMP_EOF2. XN should not be changed between calls to COMP_EOF2.

On exit, XN contains the number of observations in the data matrix.

FAILURE (OUTPUT) logical(lgl) On exit when LAST=true:

- FAILURE = false: indicates successful exit.
- FAILURE = true: indicates that fewer than two valid observations were present or that the observations on some variable were constant and the correlations were requested or that maximum accuracy was not achieved when computing the eigenvalues or that some eigenvectors failed to converge with MAXITER inverse iterations.

On exit when LAST=false, FAILURE is always set to false.

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows, if:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

COV (INPUT, OPTIONAL) logical(lgl) On entry, when argument COV is present, COV is used as follows, if:

- COV= true, the eigenvalues and the eigenvectors are computed from the variances-covariances matrix, when LAST=true.
- COV= false, the eigenvalues and the eigenvectors are computed from the correlation matrix, when LAST=true.

By default, the eigenvalues and eigenvectors of the correlation matrix are computed.

COV needs to be specified only on the last call to COMP_EOF2 (e.g., when LAST=true).

SAVECOR (INPUT, OPTIONAL) logical(lgl) On exit, when argument SAVECOR is present and LAST=true, SAVECOR is used as follows, if:

- SAVECOR= true, the correlation (or covariance) matrix is saved in packed form in argument XCORP.
- SAVECOR= false, the correlation (or covariance) matrix is destroyed.

By default, the correlation (or covariance) matrix is destroyed.

SAVECOR needs to be specified only on the last call to COMP_EOF2 (e.g., when LAST=true).

MAXITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed in the subroutine for computing the eigenvectors. By default, 2 inverse iterations are performed for all the eigenvectors. This optional argument is used only if the optional argument XEIGVEC is present.

MAXITER needs to be specified only on the last call to COMP_EOF2 (e.g., when LAST=true).

ORTHO (INPUT, OPTIONAL) logical(lgl) If ORTHO=true the computed eigenvectors are orthogonalized by the Modified Gram-Schmidt algorithm. This optional argument is used only if the optional argument XEIGVEC is present.

The default is FALSE.

ORTHO needs to be specified only on the last call to COMP_EOF2 (e.g., when LAST=true).

XMEAN (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XMEAN is present, XMEAN contains the variable means.

XMEAN needs to be specified only on the last call to COMP_EOF2 (e.g., when LAST=true).

The size of XMEAN must verify: size(XMEAN) = size(X, DIMVAR) .

XSTD (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XSTD is present, XSTD contains the variable standard-deviations.

XSTD needs to be specified only on the last call to COMP_EOF2 (e.g., when LAST=true).

The size of XSTD must verify: size(XSTD) = size(X, DIMVAR).

XEIGVAR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XEIGVAR is present, XEIGVAR contains percentages of total variance associated with the eigenvectors in the order of the eigenvalues stored in XEIGVAL.

XEIGVAR needs to be specified only on the last call to COMP_EOF2 (e.g., when LAST=true).

The size of XEIGVAR must verify: size(XEIGVAR) = size(X, DIMVAR).

XEIGVEC (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, when LAST=true, XEIGVEC contains the first size(XEIGVEC,2) eigenvectors of the variance-covariance (or correlation) matrix from the data matrix.

XEIGVEC needs to be specified only on the last call to COMP_EOF2 (e.g., when LAST=true).

The shape of XEIGVEC must verify:

- size(XEIGVEC, 1) = size(X, DIMVAR) ,
- size(XEIGVEC, 2) <= size(X, DIMVAR) .

Further Details

The subroutine computes the means and the covariance (or correlation) matrix with only one pass through the data.

On exit, if SAVECOR= true, the upper triangle of the symmetric correlation or variance-covariance matrix XCOR is packed columnwise in the linear array XCORP. More precisely, the j-th column of XCOR is stored in the array XCORP as follows:

XCORP(i + (j-1) * j/2) = XCOR(i,j) for 1<=i<=j;

Eigenvalues and selected eigenvectors are computed from the packed correlation matrix when LAST=true.

This subroutine may be used in a call with no observations (e.g., size(X,3-DIMVAR) = 0), in order to finish the computations with LAST=true when the total number of observations is unknown at the beginning of the computations.

If fewer than two valid observations were present for some pair of variables or if the observations on some variable were constants, the statistics XEIGVAL, XEIGVEC, XEIGVAR and XCORP are globally set to NaN code.

For more details on EOF or PCA analysis, see:

(1) **Jackson, J.E., 2003:** A user's guide to principal components. John Wiley and Sons, New York, USA, 592 pp., ISBN:978-0-471-47134-9

(2) **Jolliffe, I.T., 2002:** Principal component analysis. Springer-Verlag, New York, USA, 487 pp., 2nd Ed, ISBN:978-0-387-22440-4

(3) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 13, 484 pp., ISBN:9780521012300

6.12.29 subroutine comp_eof3 (x, dimvar, failure, xcorp, xeigval, xeigvec, maxiter, ortho)

Purpose

COMP_EOF3 computes estimates of Empirical Orthogonal Functions (EOF; also known as Principal Component Analysis) from a data matrix X with n observations.

COMP_EOF3 computes the matrix product (1/n) (X' * X) or (1/n) (X * X') from the data matrix X, all the eigenvalues, and selected eigenvectors (by inverse iteration), of this matrix product.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input data matrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables for which Empirical Orthogonal Functions or Principal Components are desired.

DIMVAR (INPUT) integer(i4b) On entry, DIMVAR is used as follows, if:

- DIMVAR = 1, the input matrix X contains size(X,2) observations on size(X,1) variables and the matrix product (1/n) (X * X') is computed.
- DIMVAR = 2, the input matrix X contains size(X,1) observations on size(X,2) variables and the matrix product (1/n) (X' * X) is computed.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false: indicates successful exit.
- FAILURE = true: indicates that fewer than two valid observations were present or that maximum accuracy was not achieved when computing the eigenvalues or that some eigenvectors failed to converge with MAXITER inverse iterations.

XCORP (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, XCORP(:) contains the matrix product (1/n) (X' * X) or (1/n) (X * X'), stored in symmetric storage mode.

The upper triangle of the symmetric matrix product matrix is packed columnwise in the linear array XCORP(:). More precisely, the j-th column of this matrix is stored in the array XCORP(:) as follows:

XCORP(i + (j-1) * j/2,1) = XCOR(i,j) for 1<=i<=j;

The size of XCORP must verify: size(XCORP) = (size(X,DIMVAR) * (size(X,DIMVAR)+1))/2

XEIGVAL (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit:

- XEIGVAL(:,1) contains the eigenvalues in decreasing order of the matrix product (1/n) (X' * X) or (1/n) (X * X') from the data matrix X. The near zero eigenvalues made negative by round off errors are set to zero.
- XEIGVAL(:,2) contains percentages of total variance associated with the eigenvectors in the order of the eigenvalues stored in XEIGVAL(:,1).

The shape of XEIGVAL must verify:

- size(XEIGVAL, 1) = size(X, DIMVAR) ,
- size(XEIGVAL, 2) = 2 .

XEIGVEC (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, XEIGVEC contains the first size(XEIGVEC,2) eigenvectors of the matrix product (1/n) (X' * X) or (1/n) (X * X') from the data matrix X.

The shape of XEIGVEC must verify:

- size(XEIGVEC, 1) = size(X, DIMVAR) ,
- size(XEIGVEC, 2) <= size(X, DIMVAR) .

MAXITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed in the subroutine for computing the eigenvectors. By default, 2 inverse iterations are performed for all the eigenvectors. This optional argument is used only if the XEIGVEC is present.

ORTHO (INPUT, OPTIONAL) logical(lgl) If ORTHO=true the computed eigenvectors are orthogonalized by the Modified Gram-Schmidt algorithm. This optional argument is used only if the XEIGVEC is present.

The default is FALSE.

Further Details

The subroutine computes the Empirical Orthogonal Functions or the Principal Components with only one pass through the data.

If size(X,3-DIMVAR)<=0, the subroutine set FAILURE to true and returns without doing any computations.

For more details on EOF or PCA analysis, see:

(1) **Jackson, J.E., 2003:** A user's guide to principal components. John Wiley and Sons, New York, USA, 592 pp., ISBN:978-0-471-47134-9

(2) **Jolliffe, I.T., 2002:** Principal component analysis. Springer-Verlag, New York, USA, 487 pp., 2nd Ed, ISBN:978-0-387-22440-4

(3) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 13, 484 pp., ISBN:9780521012300

6.12.30 subroutine comp_eof_miss (x, first, last, xeigval, xeigvec, xcorp, xmiss, failure, dimvar, cov, sort, maxiter, xmean, xstd)

Purpose

COMP_EOF_MISS computes estimates of Empirical Orthogonal Functions (EOF; also known as Principal Component Analysis) from a data matrix possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which Empirical Orthogonal Functions are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrix is the first submatrix of the data matrix.
- FIRST = false the current submatrix is not the first submatrix of the data matrix.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrix is the last submatrix of the data matrix.
- LAST = false the current submatrix is not the last submatrix of the data matrix.

XEIGVAL (INPUT/OUTPUT) real(stnd), dimension(:,2) On entry, after the first call to COMP_EOF_MISS (e.g., when FIRST=true), XEIGVAL contains temporary results from previous calls to COMP_EOF_MISS. XEIGVAL should not be changed between calls to COMP_EOF_MISS. On exit, when LAST=true:

- XEIGVAL(:,1) contains the eigenvalues of the variance-covariance (or correlation) matrix from the data matrix. The near zero eigenvalues made negative by round off errors or because the variance-covariance (or correlation) matrix from the data matrix with missing values is not positive definite are set to zero.
- XEIGVAL(:,2) contains percentages of total variance associated with the eigenvectors in the order of the eigenvalues stored in XEIGVAL(:,1).

The shape of XEIGVAL must verify:

- size(XEIGVAL, 1) = size(X, DIMVAR) ,
- size(XEIGVAL, 2) = 2 .

XEIGVEC (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_EOF_MISS (e.g., when FIRST=true), the matrix XEIGVEC contains temporary results from previous calls to COMP_EOF_MISS. XEIGVEC should not be changed between calls to COMP_EOF_MISS.

On exit, when LAST=true, XEIGVEC contains the eigenvectors of th variance-covariance (or correlation) matrix from the data matrix.

The shape of XEIGVEC must verify:

- size(XEIGVEC, 1) = size(X, DIMVAR) ,
- size(XEIGVEC, 2) = size(X, DIMVAR) .

XCORP (INPUT/OUTPUT) real(stnd), dimension(:,3) On entry, after the first call to COMP_EOF_MISS (e.g., when FIRST=true), XCORP is used as workspace to accumulate quantities from previous calls to COMP_EOF_MISS. XCORP should not be changed between calls to COMP_EOF_MISS.

On exit, when LAST=true:

- XCORP(:,1) contains the correlation or variance-covariance matrix, as controlled by the COV argument, stored in symmetric storage mode. The upper triangle of the symmetric correlation or variance-covariance matrix is packed columnwise in the linear array XCORP(:,1). More precisely, the j-th column of this matrix is stored in the array XCORP(:,1) as follows:

  XCORP(i + (j-1) * j/2,1) = XCOR(i,j) for 1<=i<=j;

- XCORP(:,2) contains the upper triangle of the matrix of the incidence values between each pair of variables, packed columnwise, in a linear array. XCORP(i + (j-1) * j/2,2) indicates the numbers of non-missing pairs which were used in the calculation of the covariance (or correlation) between variables i and j, for 1<=i<=j .
- XCORP(:,3) is used as workspace and contains no useful information.

The shape of XCORP must verify:

- size(XCORP, 1) = (size(X,DIMVAR) * (size(X,DIMVAR)+1))/2 ,
- size(XCORP, 2) = 3 .

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing or invalid. The means, standard-deviations and the correlations are computed on all the observations where X are not missing (see Further Details).

FAILURE (OUTPUT) logical(lgl) On exit when LAST=true:

- FAILURE = false: indicates successful exit.
- FAILURE = true: indicates that fewer than two valid observations were present for some pair of variables or that the observations on some variable were constant and the correlations were requested or that maximum accuracy was not achieved when computing the eigenvectors and the eigenvalues.

On exit when LAST=false, FAILURE is always set to false.

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows, if:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

COV (INPUT, OPTIONAL) logical(lgl) On entry, when argument COV is present, COV is used as follows, if:

- COV= true, XEIGVEC contains the eigenvectors of the variances-covariances matrix, when LAST=true.
- COV= false, XEIGVEC contains the eigenvectors of the correlation matrix, when LAST=true.

By default, the eigenvectors of the correlation matrix are output.

COV needs to be specified only on the last call to COMP_EOF_MISS (e.g., when LAST=true).

SORT (INPUT, OPTIONAL) character Sort the eigenvalues into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The eigenvectors are reordered accordingly.

SORT needs to be specified only on the last call to COMP_EOF_MISS (e.g., when LAST=true).

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of an intermediate tridiagonal form T of the covariance matrix. The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(XEIGVAL). Convergence usually occurs in about 2 * size(XEIGVAL) QR sweeps.

The default is 30.

MAXITER needs to be specified only on the last call to COMP_EOF_MISS (e.g., when LAST=true).

XMEAN (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XMEAN is present, XMEAN contains the variable means computed from all non-missing observations in the data matrix.

XMEAN needs to be specified only on the last call to COMP_EOF_MISS (e.g., when LAST=true).

The size of XMEAN must verify: size(XMEAN) = size(X, DIMVAR).

XSTD (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XSTD is present, XSTD contains the variable standard-deviations.

XSTD needs to be specified only on the last call to COMP_EOF_MISS (e.g., when LAST=true).

The size of XSTD must verify: size(XSTD) = size(X, DIMVAR).

Further Details

The subroutine computes the Empirical Orthogonal Functions with only one pass through the data.

If fewer than two valid observations were present for some pair of variables or if the observations on some variable were constants, the statistics XEIGVAL, XEIGVEC, and XCORP(:,1) are globally set to XMISS.

The means and standard-deviations of the data matrix are computed from all valid data. The correlation coefficients are based on these univariate statistics and on all valid pairs of observations. The eigenvectors and eigenvalues are computed from these bivariate statistics.

This subroutine may be used in a call with no observations (e.g., size(X,3-DIMVAR) = 0) in order to finish the computations with LAST=true when the total number of observations is unknown at the beginning of the computations.

For more details on EOF or PCA analysis, see:

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 13, 484 pp., ISBN:9780521012300

6.12.31 subroutine comp_eof_miss2 (x, first, last, xeigval, xcorp, xmiss, failure, dimvar, cov, maxiter, ortho, xmean, xstd, xeigvec)

Purpose

COMP_EOF_MISS2 computes estimates of Empirical Orthogonal Functions (EOF; also known as Principal Component Analysis) from a data matrix possibly containing missing values.

COMP_EOF_MISS2 computes all the eigenvalues, and optionally selected eigenvectors (by inverse iteration), of the covariance (or correlation matrix) from a data matrix.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which Empirical Orthogonal Functions are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrix is the first submatrix of the data matrix.
- FIRST = false the current submatrix is not the first submatrix of the data matrix.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrix is the last submatrix of the data matrix.
- LAST = false the current submatrix is not the last submatrix of the data matrix.

XEIGVAL (INPUT/OUTPUT) real(stnd), dimension(:,2) On entry, after the first call to COMP_EOF_MISS2 (e.g., when FIRST=true), XEIGVAL contains temporary results from previous calls to COMP_EOF_MISS2. XEIGVAL should not be changed between calls to COMP_EOF_MISS2.

On exit, when LAST=true:

- XEIGVAL(:,1) contains the eigenvalues of the variance-covariance (or correlation) matrix from the data matrix. The near zero eigenvalues made negative by round off errors or because the variance-covariance (or correlation) matrix from the data matrix with missing values is not positive definite are set to zero.
- XEIGVAL(:,2) contains percentages of total variance associated with the eigenvectors in the order of the eigenvalues stored in XEIGVAL(:,1).

The shape of XEIGVAL must verify:

- size(XEIGVAL, 1) = size(X, DIMVAR) ,
- size(XEIGVAL, 2) = 2 .

XCORP (INPUT/OUTPUT) real(stnd), dimension(:,4) On entry, after the first call to COMP_EOF_MISS2 (e.g., when FIRST=true), XCORP is used as workspace to accumulate quantities from previous calls to COMP_EOF_MISS2. XCORP should not be changed between calls to COMP_EOF_MISS2.

On exit:

- XCORP(:,1) contains the correlation or variance-covariance matrix, as controlled by the COV argument, stored in symmetric storage mode. The upper triangle of the symmetric correlation or variance-covariance matrix is packed columnwise in the linear array XCORP(:,1). More precisely, the j-th column of this matrix is stored in the array XCORP(:,1) as follows:

  XCORP(i + (j-1) * j/2,1) = XCOR(i,j) for 1<=i<=j;

- XCORP(:,2) contains the upper triangle of the matrix of the incidence values between each pair of variables, packed columnwise, in a linear array. XCORP(i + (j-1) * j/2,2) indicates the numbers of non-missing pairs which were used in the calculation of the covariance (or correlation) between variables i and j, for 1<=i<=j .
- XCORP(:,3:4) is used as workspace and contains no useful informations.

The shape of XCORP must verify:

- size(XCORP, 1) = (size(X,DIMVAR) * (size(X,DIMVAR)+1))/2 ,
- size(XCORP, 2) = 4 .

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing or invalid. The means, standard-deviations and the correlations are computed on all the observations where X are not missing (see Further Details).

FAILURE (OUTPUT) logical(lgl) On exit when LAST=true:

- FAILURE = false: indicates successful exit.
- FAILURE = true: indicates that fewer than two valid observations were present for some pair of variables or that the observations on some variable were constant and the correlations were requested or that maximum accuracy was not achieved when computing the eigenvalues or that some eigenvectors failed to converge with MAXITER inverse iterations.

On exit when LAST=false, FAILURE is always set to false.

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows, if:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

COV (INPUT, OPTIONAL) logical(lgl) On entry, when argument COV is present, COV is used as follows, if:

- COV= true, the eigenvalues and the eigenvectors are computed from the variances-covariances matrix, when LAST=true.
- COV= false, the eigenvalues and the eigenvectors are computed from the correlation matrix, when LAST=true.

By default, the eigenvalues and eigenvectors of the correlation matrix are computed.

COV needs to be specified only on the last call to COMP_EOF_MISS2 (e.g., when LAST=true).

MAXITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed in the subroutine for computing the eigenvectors. By default, 2 inverse iterations are performed for all the eigenvectors. This optional argument is used only if the XEIGVEC is present.

MAXITER needs to be specified only on the last call to COMP_EOF_MISS2 (e.g., when LAST=true).

ORTHO (INPUT, OPTIONAL) logical(lgl) If ORTHO=true the computed eigenvectors are orthogonalized by the Modified Gram-Schmidt algorithm. This optional argument is used only if the XEIGVEC is present.

The default is FALSE.

ORTHO needs to be specified only on the last call to COMP_EOF_MISS2 (e.g., when LAST=true).

XMEAN (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XMEAN is present, XMEAN contains the variable means computed from all non-missing observations in the data matrix.

XMEAN needs to be specified only on the last call to COMP_EOF_MISS2 (e.g., when LAST=true).

The size of XMEAN must verify: size(XMEAN) = size(X, DIMVAR).

XSTD (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XSTD is present, XSTD contains the variable standard-deviations.

XSTD needs to be specified only on the last call to COMP_EOF_MISS2 (e.g., when LAST=true).

The size of XSTD must verify: size(XSTD) = size(X, DIMVAR).

XEIGVEC (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, when LAST=true, XEIGVEC contains the first size(XEIGVEC,2) eigenvectors of the variance-covariance (or correlation) matrix from the data matrix.

XEIGVEC needs to be specified only on the last call to COMP_EOF_MISS2 (e.g., when LAST=true).

The shape of XEIGVEC must verify:

- size(XEIGVEC, 1) = size(X, DIMVAR) ,
- size(XEIGVEC, 2) <= size(X, DIMVAR) .

Further Details

The subroutine computes the Empirical Orthogonal Functions with only one pass through the data.

If fewer than two valid observations were present for some pair of variables or if the observations on some variable were constants, the statistics XEIGVAL, XEIGVEC, and XCORP(:,1) are globally set to XMISS .

The means and standard-deviations of the data matrix are computed from all valid data. The correlation coefficients are based on these univariate statistics and on all valid pairs of observations. The eigenvectors and eigenvalues are computed from these bivariate statistics.

This subroutine may be used in a call with no observations (e.g., size(X,3-DIMVAR) = 0) in order to finish the computations with LAST=true when the total number of observations is unknown at the beginning of the computations.

For more details on EOF or PCA analysis, see:

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 13, 484 pp., ISBN:9780521012300

6.12.32 subroutine comp_eof_miss3 (x, xmiss, dimvar, failure, xcorp, xincp, xeigval, xeigvec, maxiter, ortho)

Purpose

COMP_EOF_MISS3 computes estimates of Empirical Orthogonal Functions (EOF) or Principal Components (PC) from a data matrix X with n observations possibly containing missing values.

COMP_EOF_MISS3 computes an estimate of the matrix product (1/n) (X' * X) or (1/n) (X * X') from the data matrix X, the associated matrix of incidence values, all the eigenvalues, and selected eigenvectors (by inverse iteration), of this estimate of the matrix product.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input data matrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables for which Empirical Orthogonal Functions or Principal Components are desired.

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing or invalid. The estimate of the matrix product (1/n) (X' * X) or (1/n) (X * X') is computed on all the observations where X are not missing (see Further Details).

DIMVAR (INPUT) integer(i4b) On entry, DIMVAR is used as follows, if:

- DIMVAR = 1, the input matrix X contains size(X,2) observations on size(X,1) variables and the matrix product (1/n) (X * X') is computed.
- DIMVAR = 2, the input matrix X contains size(X,1) observations on size(X,2) variables and the matrix product (1/n) (X' * X) is computed.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false: indicates successful exit.
- FAILURE = true: indicates that fewer than one valid observation were present for some pair of variables or that maximum accuracy was not achieved when computing the eigenvalues or that some eigenvectors failed to converge with MAXITER inverse iterations.

XCORP (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, XCORP(:) contains the estimate of the matrix product (1/n) (X' * X) or (1/n) (X * X'), stored in symmetric storage mode.

The upper triangle of the symmetric matrix product matrix is packed columnwise in the linear array XCORP(:). More precisely, the j-th column of this matrix is stored in the array XCORP(:) as follows:

XCORP(i + (j-1) * j/2,1) = XCOR(i,j) for 1<=i<=j;

The size of XCORP must verify: size(XCORP) = (size(X,DIMVAR) * (size(X,DIMVAR)+1))/2

XINCP (OUTPUT, OPTIONAL) integer(i4b), dimension(:) On exit, XINCP(:) contains the upper triangle of the matrix of the incidence values between each pair of variables, packed columnwise, in a linear array. XINCP(i + (j-1) * j/2) indicates the numbers of non-missing pairs which were used in the calculation of the scalar product between variables i and j, for 1<=i<=j .

The size of XINCP must verify: size(XINCP) = (size(X,DIMVAR) * (size(X,DIMVAR)+1))/2

XEIGVAL (OUTPUT, OPTIONAL) real(stnd), dimension(:,2) On exit:

- XEIGVAL(:,1) contains the eigenvalues in decreasing order of the estimate of the matrix product (1/n) (X' * X) or (1/n) (X * X') from the data matrix X. The near zero eigenvalues made negative by round off errors or because the matrix product from the data matrix X with missing values is not positive definite are set to zero.
- XEIGVAL(:,2) contains percentages of total variance associated with the eigenvectors in the order of the eigenvalues stored in XEIGVAL(:,1).

The shape of XEIGVAL must verify:

- size(XEIGVAL, 1) = size(X, DIMVAR) ,
- size(XEIGVAL, 2) = 2 .

XEIGVEC (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, XEIGVEC contains the first size(XEIGVEC,2) eigenvectors of the estimate of the matrix product (1/n) (X' * X) or (1/n) (X * X') from the data matrix X.

The shape of XEIGVEC must verify:

- size(XEIGVEC, 1) = size(X, DIMVAR) ,
- size(XEIGVEC, 2) <= size(X, DIMVAR) .

MAXITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed in the subroutine for computing the eigenvectors. By default, 2 inverse iterations are performed for all the eigenvectors. This optional argument is used only if the XEIGVEC is present.

ORTHO (INPUT, OPTIONAL) logical(lgl) If ORTHO=true the computed eigenvectors are orthogonalized by the Modified Gram-Schmidt algorithm. This optional argument is used only if the XEIGVEC is present.

The default is FALSE.

Further Details

The subroutine computes the Empirical Orthogonal Functions or the Principal Components with only one pass through the data.

The estimate of the matrix product (1/n) (X' * X) or (1/n) (X * X') is computed from all valid pairs of observations. The eigenvectors and eigenvalues are computed from these bivariate statistics.

If fewer than one valid observation is present for some pair of variables, the scalar product between this pair of variables is set to zero for computing the eigenvectors and eigenvalues of the estimated matrix product.

If size(X,3-DIMVAR)<=0, the subroutine set FAILURE to true and returns without doing any computations.

For more details on EOF or PCA analysis, see:

(1) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 13, 484 pp., ISBN:9780521012300

6.12.33 subroutine comp_pc_eof (x, xeigvec, xsingval, xpc, dimvar, xmean, xstd, xpccor)

Purpose

COMP_PC_EOF computes estimates of Principal Components (PC) from a data matrix and a set of eigenvectors derived from an EOF or PCA analysis.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which Principal Components are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix.

XEIGVEC (INPUT) real(stnd), dimension(:,:) On entry, XEIGVEC contains selected eigenvectors of th variance-covariance (or correlation) matrix from the data matrix.

The shape of XEIGVEC must verify: size(XEIGVEC, 1) = size(X, DIMVAR).

XSINGVAL (INPUT) real(stnd), dimension(:) On entry, XSINGVAL must contain the singular values of the covariance (or correlation) matrix from the data matrix associated with the eigenvectors in XEIGVEC array. The Principal Components are normalized by XSINGVAL on output (the variances of the Principal Components are equal to one).

The size of XSINGVAL must verify: size(XSINGVAL) = size(XEIGVEC, 2).

XPC (OUTPUT) real(stnd), dimension(:,:) On exit, XPC contains the normalized Principal Components derived from X and XEIGVEC.

The shape of XPC must verify:

- size(XPC, 1) = size(X, 3-DIMVAR) ,
- size(XPC, 2) = size(XEIGVEC, 2) .

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows, if:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

XMEAN (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if XMEAN is present, XMEAN contains the variable means and the Principal Components are computed from the centered data matrix X.

The size of XMEAN must verify: size(XMEAN) = size(X, DIMVAR).

XSTD (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if XSTD is present, XSTD contains the variable standard-deviations and the Principal Components are computed from the normalized data matrix X.

The size of XSTD must verify: size(XSTD) = size(X, DIMVAR).

XPCCOR (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, when XPCCOR is present, XPCCOR contains the correlations (or covariances) between the data matrix XX and the principal components (factor loadings matrix).

This optional argument may be specified in a call to COMP_PC_EOF with size(X,3-DIMVAR) = size(XPC,1) = 0 (e.g., with no observations) such as:

call comp_pc_eof(x(:nvar,1:0), xeigvec(:nvar,:neigvec), xsingval(:neigvec), &
xpc(1:0,:neigvec), xpccor=xpccor(:nvar,:neigvec))

The shape of XPCCOR must verify:

- size(XPCCOR, 1) = size(X, DIMVAR) ,
- size(XPCCOR, 2) = size(XEIGVEC, 2) .

Further Details

The subroutine computes the Principal Components with only one pass through the data.

If unnormalized PCs are desired, use argument XSINGVAL with all values set to one, however in this case, do not use argument XPCCOR.

6.12.34 subroutine comp_ortho_rot_eof (fac, rot_fac, orot, std_rot_fac, failure, knorm, maxiter, w, delta)

Purpose

COMP_ORTHO_ROT_EOF performs an orthogonal rotation of a (partial) EOF model (e.g., a factor loading matrix) using a generalized orthomax criterion, including quartimax, varimax and equamax rotation methods.

Arguments

FAC (INPUT) real(stnd), dimension(:,:) On entry, the unrotated EOF model (e.g., the input factor loading matrix). The number of EOFs or factors in the model is equal to nf=size(FAC,2) and the number of variables is equal to nv=size(FAC,1).

The shape of FAC must verify:

- size(FAC, 1) = nv >= size(FAC, 2) = nf >= 2 .

ROT_FAC (OUTPUT) real(stnd), dimension(:,:) On exit, the rotated EOF model (e.g., the rotated factor loading matrix).

The shape of ROT_FAC must verify:

- size(ROT_FAC, 1) = size(FAC, 1) = nv ,
- size(ROT_FAC, 2) = size(FAC, 2) = nf .

OROT (OUTPUT) real(stnd), dimension(:,:) On exit, the computed orthogonal rotation matrix.

The shape of OROT must verify:

- size(OROT, 1) = size(OROT, 2) = size(FAC, 2) = nf .

STD_ROT_FAC (OUTPUT) real(stnd), dimension(:) On exit, the standard-deviations accounted for by the rotated EOFs.

The size of STD_ROT_FAC must verify:

- size(STD_ROT_FAC) = size(FAC, 2) = nf.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false: indicates successful exit.
- FAILURE = true: indicates that convergence did not occur in MAXITER iterations. But, convergence was assumed and calculations continued so that the results can still be useful.

KNORM (INPUT, OPTIONAL) logical(lgl) On entry, if:

- KNORM = true indicates that the rows of the input unrotated EOF model must be normalized following Kaiser's method.
- KNORM = false indicates that row normalization is not required.

The default is KNORM=true.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of iterations allowed for rotations. MAXITER <= 10 defaults to 10 iterations.

The default is MAXITER = 30.

W (INPUT, OPTIONAL) real(stnd) Input constant factor used to define the method of rotation. W can be any positive real number, but best values lie between 0. and 5.*nf. W <= 0. defaults to 0. .

On input, if:

- W = 0. the quartimax method is used.
- W = 1. the varimax method is used.
- W = size(FAC,2)/2. the equamax method is used.

See Further Details for more information.

The default is W = 1., which means that the varimax method of rotation is used by default.

DELTA (INPUT, OPTIONAL) real(stnd) Input convergence constant for rotation (e.g., criterion function; see Further Details). When the relative change in the criterion function is less than DELTA from one iteration to the next, convergence is assumed. DELTA=0.001 is typical. DELTA <= epsilon(DELTA) defaults to epsilon(DELTA). Similarly, DELTA > 0.02 defaults to 0.02.

The default is DELTA = 0.001.

Further Details

The subroutine performs an orthogonal rotation according to a generalized orthomax criterion. In this analytic method of orthogonal rotation, a criterion function is defined as

Q = ([sum i=1 to nv][sum j=1 to nf] ROT_FAC(i,j)**4) - (W/nv).([sum j=1 to nf] ([sum i=1 to nv] ROT_FAC(i,j)**2)**2)/nv

, where W is a positive user-specified constant yielding a family of rotations, nv is the number of variables and nf is the number of factors or EOFs and this function is maximized by finding a nf-by-nf orthogonal rotation matrix OROT such that

ROT_FAC = FAC * OROT

is a maximum of Q (here FAC is the specified matrix of the unrotated EOFs, e.g., the unrotated factor loading matrix).

W is a parameter determining the kind of solution to be computed and may be set as follows:

- W = 0. is the quartimax method, which attempts to get each variable to load highly on only one (or a few) EOFs or factors.
- W = 1. is the varimax method, which attempts to load highly a relatively low number of variables on each EOF or factor. Varimax is the most widely used method of orthogonal rotation.
- W = nf/2. is the equamax method, which is a compromise of the above two.

W can be any positive real number, but best values lie in the closed interval [0., 5.*nf]. Generally, the larger W is, the more equal is the dispersion of the variance accounted for across the rotated factors.

The method for optimizing Q proceeds by accumulating simple rotations where a simple rotation is defined to be one in which Q is optimized for two columns of ROT_FAC and for which the requirement that ROT be an orthogonal matrix is satisfied. A single iteration is defined to be such that each of the nf.(nf-1) possible simple rotations is performed (where nf is the number of EOFs or factors).

When the relative change in the criterion function Q from one iteration to the next is less than DELTA (the user-specified convergence criterion), the algorithm stops. DELTA=0.001 is generaly sufficient. Alternatively, the algorithm stops when the user-specified maximum number of iterations, MAXITER, is reached. MAXITER=30 is usually sufficient.

Kaiser (row) normalization can be performed on the EOFs (e.g., the factor loadings) prior to the rotation via the optional logical paramter KNORM. If, on input KNORM=true, the rows of FAC are first "normalized" by dividing each row by the square root of the sum of its squared elements. After the rotation is complete, each row of ROT_FAC is "denormalized" by multiplication by its initial normalizing constant.

The documentation of this subroutine is partially adapted from the FROTA subroutine in the IMSL library, which performs exactly the same task.

For more details on orthogonal rotations for EOF or PCA analysis, see:

(1) **Jackson, J.E., 2003:** A user's guide to principal components. John Wiley and Sons, New York, USA, Chapter 8, 592 pp., ISBN:978-0-471-47134-9

(2) **Jolliffe, I.T., 2002:** Principal component analysis. Springer-Verlag, New York, USA, Chapters 7 and 11, 487 pp., 2nd Ed, ISBN:978-0-387-22440-4

(3) **von Storch, H., and Zwiers, F.W., 2002:** Statistical Analysis in Climate Research. Cambridge, UK, Chapter 13, 484 pp., ISBN:9780521012300

(4) **Jennrich, R.I., 1970:** Orthogonal rotation algorithms. Psychometrika, Vol. 35, 229-235

(5) **Clarkson, D.B., and Jennrich, R.I., 1988:** Quartic rotation criteria and algorithms. Psychometrika, Vol. 53, 251-259

(6) **Kaiser, H.F., 1958:** The varimax criterion for analytic rotation in factor analysis. Psychometrika, Vol. 23, 187-200

(7) **Jennrich, R.I., 2001:** A simple general procedure for orthogonal rotation. Psychometrika, Vol. 66, 289-306

(8) **Bernaards, C.A., and Jennrich, R.I., 2005:** Gradient Projection Algorithms and Software for Arbitrary Rotation Criteria in Factor Analysis. Educational and Psychological Measurement, Vol. 65, 676-696

6.12.35 subroutine comp_smooth_rot_pc (pc, std_pc, rot_pc, orot, std_rot_pc, failure, maxiter, d, smooth)

Purpose

COMP_SMOOTH_ROT_PC performs an orthogonal rotation of a (partial) EOF model (e.g., the standardized Principal Component time series) by minimizing a smoothness criterion.

Arguments

PC (INPUT) real(stnd), dimension(:,:) On entry, the unrotated EOF model (e.g., the standardized Principal Component time series). The number of EOFs or factors in the model is equal to nf=size(PC,2) and the number of observations is equal to nobs=size(PC,1). The time observations in the original dataset must be ordered, but not neccessarily equally spaced. See Further details and the description the optional real vector argument D below for more information.

The shape of PC must verify:

- size(PC, 1) = nobs >= size(PC, 2) = nf >= 2 .

STD_PC (INPUT) real(stnd), dimension(:) On entry, the standard-deviations accounted for by the input standardized Principal Component time series.

The size of STD_PC must verify:

- size(STD_PC) = size(PC, 2) = nf.

ROT_PC (OUTPUT) real(stnd), dimension(:,:) On exit, the rotated EOF model (e.g., the rotated standardized Principal Component time series).

The shape of ROT_PC must verify:

- size(ROT_PC, 1) = size(PC, 1) = nobs ,
- size(ROT_PC, 2) = size(PC, 2) = nf .

OROT (OUTPUT) real(stnd), dimension(:,:) On exit, the computed orthogonal rotation matrix.

The shape of OROT must verify:

- size(OROT, 1) = size(OROT, 2) = size(PC, 2) = nf .

STD_ROT_PC (OUTPUT) real(stnd), dimension(:) On entry, the standard-deviations accounted for by the rotated standardized Principal Component time series.

The size of STD_ROT_PC must verify:

- size(STD_ROT_PC) = size(PC, 2) = nf.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false: indicates successful exit.
- FAILURE = true: indicates that convergence did not occur in MAXITER iterations. But, convergence was assumed and calculations continued so that the results can still be useful.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of the symmetric matrix used to find the minima of the smoothness criterion.

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(PC,2). Convergence usually occurs in about 2 * size(PC,2) QR sweeps.

The default is 30.

D (INPUT, OPTIONAL) real(stnd), dimension(:) Input vector indexing the time observations, if the time interval between sucessive observations is not constant. This optional argument has no effect on the computed solutions if the time interval is constant.

See Further Details for more information.

The size of D must verify the relation:

- size(D) = nobs .

SMOOTH (OUTPUT, OPTIONAL) real(stnd), dimension(:) Smoothness criteria for the rotated standardized Principal Component time series.

See Further Details for more information.

The size of SMOOTH must verify the relation:

- size(SMOOTH) = nf .

Further Details

The subroutine performs an orthogonal rotation of standardized Principal Component time series according to a smoothness criterion, which is defined as

SM(Y) = || Y(i) - Y(i-1) ||**2 = [sum i=2 to nobs] (Y(i) - Y(i-1))**2

for a vector Y of nobs observations and satisfying || Y || = 1 (or Var(Y) = 1). More precisely, the subroutine finds the linear combinations of the input Principal Component time series, which give successively the minimum of SM(Y) with the requirement that the rotated Principal Component time series remain orthogonal (e.g., uncorrelated) and of norm unity. This is equivalent to find a nf-by-nf orthogonal rotation matrix OROT such that

ROT_PC = PC * OROT

and [sum i=1 to nf] SM(ROT_PC(:,i)) is a minimum.

The smoothness criterion weights each squared successive difference equally, based on the assumption that the time observations are ordered and equally spaced. If the observations are not taken in successive, equally spaced time periods, the smoothness criterion can be suitably modified as

SM(Y) = [sum i=2 to nobs] ((Y(i) - Y(i-1))/(D(i) - D(i-1)))**2

where the D vector gives the spacing between observations, and so that each squared successive difference is weighted accordingly to the unequal spacing between observations when computing the smoothness criterion.

For more details on orthogonal rotations to smooth functions, see:

(1) **Arbuckle, J., and Friendly, M.L., 1977:** On rotating to smooth functions. Psychometrika, Vol. 42, 127-140

(2) **Solow, A.R., and Patwardhan, A., 1996:** Extracting a smooth trend from a time series: A modification of Singular Spectrum Analysis. Journal of Climate, Vol. 9, 2163-2166

(3) **Jolliffe, I.T., 2002:** Principal component analysis. Springer-Verlag, New York, USA, Chapters 7 and 11, 487 pp., 2nd Ed, ISBN:978-0-387-22440-4

6.12.36 subroutine comp_lfc_rot_pc (pc, std_pc, nt, rot_pc, orot, std_rot_pc, failure, maxiter, itdeg, ntjump, residual, smooth)

Purpose

COMP_LFC_ROT_PC performs an orthogonal rotation of a (partial) EOF model (e.g., the standardized Principal Component time series) towards low-frequency or high-frequency components using the eigenvectors of the covariance matrix between the standardized Principal Component time series filtered with a LOESS smoother.

Arguments

PC (INPUT) real(stnd), dimension(:,:) On entry, the unrotated EOF model (e.g., the standardized Principal Component time series). The number of EOFs or factors in the model is equal to nf=size(PC,2) and the number of observations is equal to nobs=size(PC,1). The time observations in the original dataset must be ordered and equally spaced.

The shape of PC must verify:

- size(PC, 1) = nobs >= size(PC, 2) = nf >= 2 .

STD_PC (INPUT) real(stnd), dimension(:) On entry, the standard-deviations accounted for by the input standardized Principal Component time series.

The size of STD_PC must verify:

- size(STD_PC) = size(PC, 2) = nf.

NT (INPUT) integer(i4b) On entry, the length of the LOESS trend smoother. The value of NT should be an odd integer greater than or equal to 3. As NT increases the values of the rotated standardized PC components become smoother if residual=false or rougher (e.g., high-frequency) if residual=true.

ROT_PC (OUTPUT) real(stnd), dimension(:,:) On exit, the rotated EOF model (e.g., the rotated standardized Principal Component time series).

The shape of ROT_PC must verify:

- size(ROT_PC, 1) = size(PC, 1) = nobs ,
- size(ROT_PC, 2) = size(PC, 2) = nf .

OROT (OUTPUT) real(stnd), dimension(:,:) On exit, the computed orthogonal rotation matrix.

The shape of OROT must verify:

- size(OROT, 1) = size(OROT, 2) = size(PC, 2) = nf .

STD_ROT_PC (OUTPUT) real(stnd), dimension(:) On entry, the standard-deviations accounted for by the rotated standardized Principal Component time series.

The size of STD_ROT_PC must verify:

- size(STD_ROT_PC) = size(PC, 2) = nf.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false: indicates successful exit.

- FAILURE = true: indicates that convergence did not occur in MAXITER iterations. But, convergence was assumed and calculations continued so that the results can still be useful.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of the covariance matrix used to find the orthogonal rotation matrix OROT.

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(PC,2). Convergence usually occurs in about 2 * size(PC,2) QR sweeps.

The default is 30.

ITDEG (INPUT, OPTIONAL) integer(i4b) On entry, the degree of locally-fitted polynomial in LOESS trend smoothing. The value must be 0, 1 or 2.

By default, ITDEG is set to 1.

NTJUMP (INPUT, OPTIONAL) integer(i4b) On entry, the skipping value for LOESS trend smoothing.

By default, NTJUMP is set to NT/10.

RESIDUAL (INPUT, OPTIONAL) logical(lgl) On entry, if:

- RESIDUAL = true : the rotation is done towards high-frequency components, e.g., using the eigenvectors of the covariance matrix between the residual from the trends of the original standardized Principal Component time series.
- RESIDUAL = false : the rotation is done towards low-frequency components, e.g., using the eigenvectors of the covariance matrix between the trends of the original standardized Principal Component time series.

In both cases, the trends are estimated with a LOESS smoother determined by the values of the NT, ITDEG and NTJUMP arguments.

By default, RESIDUAL = false.

SMOOTH (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a vector containing the ratios of filtered variance to total variance for the rotated standardized Principal Component time series.

See Further Details for more information.

The size of SMOOTH must verify the relation:

- size(SMOOTH) = nf .

Further Details

The subroutine performs an orthogonal rotation of standardized Principal Component time series towards low- or high-frequency components using a framework described in the reference (1) as Low-Frequency Components Analysis (LFCA).

Here, LFCA is considered as an orthogonal rotation of the original Principal Component time series of a (partial) EOF model. The nf-by-nf orthogonal matrix used in the rotation is computed as the eigenvectors of the covariance (e.g., symmetric positive-definite) matrix between the filtered original standardized Principal Component time series. The filtering of the Principal Component time series is performed with the help of a LOESS smoother specified by the values of the NT, ITDEG and NTJUMP arguments.

Depending on the value of the optional logical argument RESIDUAL, the filtered standardized Principal Component time series are computed as residuals from the trends estimated with the LOESS smoother (if RESIDUAL = true) or as the trends estimated with the LOESS smoother (if RESIDUAL = false).

In the first case, the rotated standardized Principal Component time series will be ordered from high-frequency to low-frequency modes and in the second case, they will be ordered from low-frequency to high-frequency modes.

For more details on orthogonal rotations to smooth functions, LFCA or LOESS smoothing, see:

(1) **Wills, R.C., Schneider, T., Wallace, J.M., Battisti, D.S., and Hartmann, D.L., 2018:** Disentangling Global Warming, Multidecadal Variability, and El Nino in Pacific Temperatures. Geophysical Research Letters, Vol. 45, 2487-2496

(2) **Cleveland, W.S., 1979:** Robust Locally Weighted Regression and Smoothing Scatterplots. Journal of the American Statistical Association, Vol. 74, 829-836

(3) **Cleveland, W.S., and Devlin, S.J., 1988:** Locally Weighted Regression: An Approach to Regression Analysis by Local Fitting. Journal of the American Statistical Association, Vol. 83, 596-610

(4) **Cleveland, R.B., Cleveland, W.S., McRae, J.E., and Terpenning, I., 1990:** STL: A Seasonal-Trend Decomposition Procedure Based on Loess. J. Official Stat., 6, 3-73

6.12.37 subroutine comp_filt_rot_pc (pc, std_pc, pl, ph, rot_pc, orot, std_rot_pc, failure, maxiter, trend, win, smooth)

Purpose

COMP_FILT_ROT_PC performs an orthogonal rotation of a (partial) EOF model (e.g., the standardized Principal Component time series) towards low-frequency, high-frequency or band-pass components using the eigenvectors of the covariance matrix between the standardized Principal Component time series filtered with a windowed FFT filter.

Arguments

PC (INPUT) real(stnd), dimension(:,:) On entry, the unrotated EOF model (e.g., the standardized Principal Component time series). The number of EOFs or factors in the model is equal to nf=size(PC,2) and the number of observations is equal to nobs=size(PC,1). The time observations in the original dataset must be ordered and equally spaced.

The shape of PC must verify:

- size(PC, 1) = nobs >= size(PC, 2) = nf >= 2 .

STD_PC (INPUT) real(stnd), dimension(:) On entry, the standard-deviations accounted for by the input standardized Principal Component time series.

The size of STD_PC must verify:

- size(STD_PC) = size(PC, 2) = nf.

PL (INPUT) integer(i4b) Minimum period of oscillation of desired components in number of timesteps for the filtered standardized PC time series used for computing the orthogonal rotation matrix.

Use PL=0 for high-pass filtering frequencies corresponding to periods shorter than PH,

PL must be equal to 0 or greater or equal to 2. Moreover, PL must be less or equal to nobs.

PH (INPUT) integer(i4b) Maximum period of oscillation of desired components in number of timesteps for the filtered standardized PC time series used for computing the orthogonal rotation matrix.

USE PH=0 for low-pass filtering frequencies corresponding to periods longer than PL.

PH must be equal to 0 or greater or equal to 2. Moreover, PH must be less or equal to nobs.

ROT_PC (OUTPUT) real(stnd), dimension(:,:) On exit, the rotated EOF model (e.g., the rotated standardized Principal Component time series).

The shape of ROT_PC must verify:

- size(ROT_PC, 1) = size(PC, 1) = nobs ,
- size(ROT_PC, 2) = size(PC, 2) = nf .

OROT (OUTPUT) real(stnd), dimension(:,:) On exit, the computed orthogonal rotation matrix.

The shape of OROT must verify:

- size(OROT, 1) = size(OROT, 2) = size(PC, 2) = nf .

STD_ROT_PC (OUTPUT) real(stnd), dimension(:) On entry, the standard-deviations accounted for by the rotated standardized Principal Component time series.

The size of STD_ROT_PC must verify:

- size(STD_ROT_PC) = size(PC, 2) = nf.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false: indicates successful exit.
- FAILURE = true: indicates that convergence did not occur in MAXITER iterations. But, convergence was assumed and calculations continued so that the results can still be useful.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the Schur decomposition of the covariance matrix used to find the orthogonal rotation matrix OROT.

The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * size(PC,2). Convergence usually occurs in about 2 * size(PC,2) QR sweeps.

The default is 30.

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+/-1 The means of the PC time series are removed before time filtering
- TREND=+/-2 The drifts from the PC time series are removed before time filtering by using the formula: drift(:) = (PC(nobs,:nf) - PC(1,:nf))/(nobs - 1)
- TREND=+/-3 The least-squares lines from the PC time series are removed before time filtering.

IF TREND=-1,-2 or -3, the means, drifts or least-squares lines are reintroduced post-filtering, respectively.

For other values of TREND nothing is done before or after filtering the standardized Principal Component time series.

WIN (INPUT, OPTIONAL) real(stnd) By default, Hamming window filtering is used (i.e. WIN=0.54). SET WIN=0.5 for Hanning window or WIN=1 for rectangular window.

WIN must be greater or equal to O.5 and less or equal to 1.

SMOOTH (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a vector containing the ratios of filtered variance to total variance for the rotated standardized Principal Component time series.

The size of SMOOTH must verify the relation:

- size(SMOOTH) = nf .

Further Details

The subroutine performs an orthogonal rotation of standardized Principal Component time series towards low-, high-frequency or band-pass components using a framework described in the reference (1) as Low-Frequency Components Analysis (LFCA).

Here, LFCA is considered as an orthogonal rotation of the original Principal Component time series of a (partial) EOF model. The nf-by-nf orthogonal matrix used in the rotation is computed as the eigenvectors of the covariance (e.g., symmetric positive-definite) matrix between the filtered original standardized Principal Component time series. The filtering of the Principal Component time series is performed with the help of a windowed FFT filter specified by the values of the PL, PH, TREND and WIN arguments. See reference (2) for more detailed on the windowed FFT filter used here. This windowed filter is also implemented in subroutine HWFILTER, which is included in module Time_Series_Procedures.

Use PL=0 for high-pass filtering frequencies corresponding to periods shorter than PH, or PH=0 for low-pass filtering frequencies corresponding to periods longer than PL.

Setting PH<PL is also allowed and performs band rejection of periods between PH and PL (i.e. in that case the meaning of the PL and PH arguments are reversed).

Examples:

- For quarterly data, PL=6, PH=32 perform rotation towards Principal Component time series with periods between 1.5 and 8 yrs.
- For monthly data, PL=0, PH=24 perform rotation towards Principal Component time series with periods less than 2 yrs.

Thus, depending on the values of the PL and PH arguments, the rotated standardized Principal Component time series will be ordered from high-frequency to low-frequency modes or vice-versa.

For more details on orthogonal rotations to smooth functions, LFCA or windowed filtering, see:

(1) **Wills, R.C., Schneider, T., Wallace, J.M., Battisti, D.S., and Hartmann, D.L., 2018:** Disentangling Global Warming, Multidecadal Variability, and El Nino in Pacific Temperatures. Geophysical Research Letters, Vol. 45, 2487-2496

(2) **Iacobucci, A., and Noullez, A., 2005:** A Frequency Selective Filter for Short-Length Time Series. Computational Economics, 25,75-102.

6.12.38 subroutine comp_mca (x, y, first, last, xstat, ystat, xysingval, xsingvec, failure, dimvarx, dimvary, cov, sort, maxiter, ysingvec, xysingvar, xycor)

Purpose

COMP_MCA performs Maximum Covariance Analysis (MCA) or canonical covariance analysis between two data matrices XX and YY.

COMP_MCA computes the singular value decomposition (SVD) of the correlation (or covariance) matrix XYCOR between two data matrices XX and YY. This SVD is written

XYCOR = U * SIGMA * V'

where SIGMA is a m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is a m-by-m orthogonal matrix, and V is a n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of XYCOR; they are real and non-negative. The first min(m,n) columns of U and V are the left and right singular vectors of XYCOR.

The routine returns the singular values, the left and, optionally, the right singular vectors of the correlation (or covariance) matrix XYCOR between two data matrices XX and YY.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVARX) observations on size(X,DIMVARX) variables from the "left" matrix of data XX. By default, DIMVARX is equal to 1. See description of optional DIMVARX argument for details. If all the data are available at once, X can be the full data matrix XX.

The shape of X must verify: size(X, 3-DIMVARX) = size(Y, 3-DIMVARY) .

Y (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(Y,3-DIMVARY) observations on size(Y,DIMVARY) variables from the "right" matrix of data YY. By default, DIMVARY is equal to 1. See description of optional DIMVARY argument for details. If all the data are available at once, Y can be the full data matrix YY.

The shape of Y must verify: size(Y, 3-DIMVARY) = size(X, 3-DIMVARX) .

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrices are the first submatrices of the data matrices XX and YY.
- FIRST = false the current submatrices are not the first submatrices of the data matrices XX and YY.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrices are the last submatrices of the data matrices XX and YY.
- LAST = false the current submatrix are not the last submatrices of the data matrices XX and YY.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,2) On entry, after the first call to COMP_MCA (e.g., when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_MCA. XSTAT should not be changed between calls to COMP_MCA.

On exit, when LAST=true, XSTAT contains the following statistics on all variables from the XX matrix:

- XSTAT(:,1) contains the mean values of the "left" data matrix XX.
- XSTAT(:,2) contains the standard-deviations of the "left" data matrix XX.

The shape of XSTAT must verify:

- size(XSTAT, 1) = size(X, DIMVARX) ,
- size(XSTAT, 2) = 2 .

YSTAT (INPUT/OUTPUT) real(stnd), dimension(:,2) On entry, after the first call to COMP_MCA (e.g., when FIRST=true), YSTAT is used as workspace to accumulate quantities on previous calls to COMP_MCA. YSTAT should not be changed between calls to COMP_MCA.

On exit, when LAST=true, YSTAT contains the following statistics on all variables from the YY matrix:

- YSTAT(:,1) contains the mean values of the "right" data matrix YY.
- YSTAT(:,2) contains the the standard-deviations of the "right" data matrix YY.

The shape of YSTAT must verify:

- size(YSTAT, 1) = size(Y, DIMVARY) ,

- size(YSTAT, 2) = 2 .

XYSINGVAL (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_MCA (e.g., when FIRST=true), XYSINGVAL(1) contains count of observations from previous calls to COMP_MCA. XYSINGVAL(1) should not be changed between calls to COMP_MCA.

On exit, XYSINGVAL contains the singular values of the correlation (or covariance) matrix XYCOR between the data matrices XX and YY.

The size of XYSINGVAL must verify: size(XYSINGVAL) = min(size(X,DIMVARX), size(Y,DIMVARY)).

XSINGVEC (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_MCA (e.g., when FIRST=true), XSINGVEC is used as workspace to accumulate quantities on previous calls to COMP_MCA. XSINGVEC should not be changed between calls to COMP_MCA.

On exit, when LAST=true, XSINGVEC is overwritten with the first min(size(X,DIMVARX),size(Y,DIMVARY)) columns of U, the left singular vectors of the correlation (or covariance) matrix XYCOV between XX and YY.

The shape of XSINGVEC must verify:

- size(XSINGVEC, 1) = size(X, DIMVARX) ,
- size(XSINGVEC, 2) = size(Y, DIMVARY).

FAILURE (OUTPUT) logical(lgl) On exit when LAST=true:

- FAILURE = false: indicates successful exit.
- FAILURE = true: indicates that fewer than two valid observations were present or that maximum accuracy was not achieved when computing the SVD of the covariance (or correlation) matrix between the data matrices XX and YY .

On exit when LAST=false, FAILURE is always set to false.

DIMVARX (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVARX is present, DIMVARX is used as follows, if:

- DIMVARX = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVARX = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables, respectively.

The default is DIMVARX = 1.

DIMVARY (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVARY is present, DIMVARY is used as follows, if:

- DIMVARY = 1, the input submatrix Y contains size(Y,2) observations on size(Y,1) variables.
- DIMVARY = 2, the input submatrix Y contains size(Y,1) observations on size(Y,2) variables, respectively.

The default is DIMVARY = 1.

COV (INPUT, OPTIONAL) logical(lgl) On entry, if COV is present and COV=true, a covariance matrix between the data matrices XX and YY is computed instead of a correlation matrix.

By default, a correlation matrix is computed.

COV needs to be specified only on the last call to COMP_MCA (e.g., when LAST=true).

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The singular vectors are rearranged accordingly.

SORT needs to be specified only on the last call to COMP_MCA (e.g., when LAST=true).

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm. The bidiagonal SVD algorithm of an intermediate bidiagonal form B of XYCOR fails to converge if the number of QR sweeps exceeds MAXITER * min(m,n). Convergence usually occurs in about 2 * min(m,n) QR sweeps.

The default is 10.

MAXITER needs to be specified only on the last call to COMP_MCA (e.g., when LAST=true).

YSINGVEC (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, when LAST=true and YSINGVEC is present, YSINGVEC contains the first min(size(X,DIMVARX),size(Y,DIMVARY)) columns of V, the right singular vectors of the correlation (or covariance) matrix XYCOR between XX and YY.

YSINGVEC needs to be specified only on the last call to COMP_MCA (e.g., when LAST=true).

The shape of YSINGVEC must verify:

- size(YSINGVEC, 1) = size(Y, DIMVARY) ,
- size(YSINGVEC, 2) = min(size(X,DIMVARX), size(Y,DIMVARY)) .

XYSINGVAR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XYSINGVAR is present, XYSINGVAR contains percentages of total squared covariance associated with the left and right singular vectors in order of the singular values stored in XYSINGVAL.

XYSINGVAR needs to be specified only on the last call to COMP_MCA (e.g., when LAST=true).

The size of XYSINGVAR must verify: size(XYSINGVAR) = min(size(X,DIMVARX), size(Y,DIMVARY)).

XYCOR (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, when LAST=true and XYCOR is present, XYCOR contains the correlation or variance-covariance matrix between data arrays XX and YY, as controlled by the COV argument.

XYCOR needs to be specified only on the last call to COMP_MCA (e.g., when LAST=true).

The shape of XYCOR must verify:

- size(XYCOR, 1) = size(X, DIMVARX) ,
- size(XYCOR, 2) = size(Y, DIMVARY) .

Further Details

The subroutine computes the basic univariate statistics and the correlation (or covariance) matrix with only one pass through the data.

If fewer than two valid observations were present, the statistics XSTAT, YSTAT, XYSINGVAL XSINGVEC, YSINGVEC, XYSINGVAR and XYCOR are set to Nan code.

This subroutine may be used in a call with no observations (e.g., size(X,3-DIMVARX) = 0) in order to finish the computations with LAST=true when the total number of observations is unknown at the beginning of the computations.

6.12.39 subroutine comp_mca2 (x, y, first, last, xstat, ystat, xysingval, xycor, failure, dimvarx, dimvary, cov, savecor, maxiter, ortho, xysingvar, xysingvec)

Purpose

COMP_MCA2 performs Maximum Covariance Analysis (MCA) or canonical covariance analysis between two data matrices XX and YY.

COMP_MCA2 computes a partial singular value decomposition (SVD) of the correlation (or covariance) matrix XYCOR between two data matrices XX and YY. This partial SVD is written

U(:m,:k) * SIGMA(:k,:k) * V(:n,:k)'

where SIGMA is a k-by-k matrix which is zero except for its k diagonal elements, U is a m-by-k orthogonal matrix, and V is a n-by-k orthogonal matrix. The diagonal elements of SIGMA are the first k singular values of XYCOR; they are real and non-negative. The k columns of U and V are the first k left and right singular vectors of XYCOR.

COMP_MCA2 computes all the singular values, and, optionally, selected left and right singular vectors (by inverse iteration), of the covariance (or correlation matrix) XYCOR between two data matrices XX and YY.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVARX) observations on size(X,DIMVARX) variables from the "left" matrix of data XX. By default, DIMVARX is equal to 1. See description of optional DIMVARX argument for details. If all the data are available at once, X can be the full data matrix XX.

The shape of X must verify: size(X, 3-DIMVARX) = size(Y, 3-DIMVARY) .

Y (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(Y,3-DIMVARY) observations on size(Y,DIMVARY) variables from the "right" matrix of data YY. By default, DIMVARY is equal to 1. See description of optional DIMVARY argument for details. If all the data are available at once, Y can be the full data matrix YY.

The shape of Y must verify: size(Y, 3-DIMVARY) = size(X, 3-DIMVARX) .

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrices are the first submatrices of the data matrices XX and YY.
- FIRST = false the current submatrices are not the first submatrices of the data matrices XX and YY.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrices are the last submatrices of the data matrices XX and YY.
- LAST = false the current submatrix are not the last submatrices of the data matrices XX and YY.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,2) On entry, after the first call to COMP_MCA (e.g., when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_MCA. XSTAT should not be changed between calls to COMP_MCA.

On exit, when LAST=true, XSTAT contains the following statistics on all variables from the XX matrix:

- XSTAT(:,1) contains the mean values of the "left" data matrix XX.

- XSTAT(:,2) contains the standard-deviations of the "left" data matrix XX.

The shape of XSTAT must verify:

- size(XSTAT, 1) = size(X, DIMVARX) ,
- size(XSTAT, 2) = 2 .

YSTAT (INPUT/OUTPUT) real(stnd), dimension(:,2) On entry, after the first call to COMP_MCA (e.g., when FIRST=true), YSTAT is used as workspace to accumulate quantities on previous calls to COMP_MCA. YSTAT should not be changed between calls to COMP_MCA.

On exit, when LAST=true, YSTAT contains the following statistics on all variables from the YY matrix:

- YSTAT(:,1) contains the mean values of the "right" data matrix YY.
- YSTAT(:,2) contains the the standard-deviations of the "right" data matrix YY.

The shape of YSTAT must verify:

- size(YSTAT, 1) = size(Y, DIMVARY) ,
- size(YSTAT, 2) = 2 .

XYSINGVAL (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_MCA2 (e.g., when FIRST=true), XYSINGVAL(1) contains count of observations from previous calls to COMP_MCA2. XYSINGVAL(1) should not be changed between calls to COMP_MCA2.

On exit, XYSINGVAL contains the singular values of the correlation (or covariance) matrix XYCOR between the data matrices XX and YY.

The size of XYSINGVAL must verify: size(XYSINGVAL) = min(size(X,DIMVARX), size(Y,DIMVARY)).

XYCOR (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_MCA2 (e.g., when FIRST=true), XYCOR is used as workspace to accumulate quantities on previous calls to COMP_MCA2. XYCOR should not be changed between calls to COMP_MCA2.

On exit, when LAST=true and SAVECOR=true, XYCOR contains the correlation or variance-covariance matrix as controlled by the COV argument. In this case XYCOR(i,j) contains the correlation (or covariance) coefficient between XX(i,:) and YY(j,:) (XX(:,i) and YY(:,j) if DIMVARX=2 and DIMVARY=2).

If SAVECOR=false, the correlation (or covariance) matrix is not saved on exit. In this case, XYCOR does not contain useful information.

The shape of XYCOR must verify:

- size(XYCOR, 1) = size(X, DIMVARX) ,
- size(XYCOR, 2) = size(Y, DIMVARY) .

FAILURE (OUTPUT) logical(lgl) On exit when LAST=true:

- FAILURE = false: indicates successful exit.
- FAILURE = true: indicates that fewer than two valid observations were present or that maximum accuracy was not achieved when computing the singular values or that some singular vectors failed to converge with MAXITER inverse iterations.

On exit when LAST=false, FAILURE is always set to false.

DIMVARX (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVARX is present, DIMVARX is used as follows, if:

- DIMVARX = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVARX = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables, respectively.

The default is DIMVARX = 1.

DIMVARY (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVARY is present, DIMVARY is used as follows, if:

- DIMVARY = 1, the input submatrix Y contains size(Y,2) observations on size(Y,1) variables.
- DIMVARY = 2, the input submatrix Y contains size(Y,1) observations on size(Y,2) variables, respectively.

The default is DIMVARY = 1.

COV (INPUT, OPTIONAL) logical(lgl) On entry, if COV is present and COV=true, a covariance matrix between the data matrices XX and YY is computed instead of a correlation matrix.

By default, a correlation matrix is computed.

COV needs to be specified only on the last call to COMP_MCA2 (e.g., when LAST=true).

SAVECOR (INPUT, OPTIONAL) logical(lgl) On exit, when argument SAVECOR is present and LAST=true, SAVECOR is used as follows, if:

- SAVECOR= true, the correlation (or covariance) matrix is saved in argument XYCOR.
- SAVECOR= false, the correlation (or covariance) matrix is destroyed.

By default, the correlation (or covariance) matrix is destroyed.

SAVECOR needs to be specified only on the last call to COMP_MCA2 (e.g., when LAST=true).

MAXITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed in the subroutine for computing the singular vectors. By default, 2 inverse iterations are performed for all the singular vectors. This optional argument is used only if the XYSINGVEC argument is present.

MAXITER needs to be specified only on the last call to COMP_MCA2 (e.g., when LAST=true).

ORTHO (INPUT, OPTIONAL) logical(lgl) If ORTHO=true the computed singular vectors are orthogonalized by the Modified Gram-Schmidt algorithm. This optional argument is used only if the XYSINGVEC argument is present.

The default is FALSE.

ORTHO needs to be specified only on the last call to COMP_MCA2 (e.g., when LAST=true).

XYSINGVAR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XYSINGVAR is present, XYSINGVAR contains percentages of total squared covariance associated with the left and right singular vectors in order of the singular values stored in XYSINGVAL.

XYSINGVAR needs to be specified only on the last call to COMP_MCA2 (e.g., when LAST=true).

The size of XYSINGVAR must verify: size(XYSINGVAR) = min(size(X,DIMVARX), size(Y,DIMVARY)).

XYSINGVEC (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, when LAST=true and XYSINGVEC is present, XYSINGVEC contains the first columns of U and V, the first k left and right singular vectors of the correlation (or covariance) matrix XYCOR between XX and YY.

The first k left singular vectors are stored in XYSINGVEC(:size(X,DIMVARX),:) The first k right singular vectors are stored in XYSINGVEC(size(X,DIMVARX)+1:,:)

XYSINGVEC needs to be specified only on the last call to COMP_MCA2 (e.g., when LAST=true).

The shape of XYSINGVEC must verify:

- size(XYSINGVEC, 1) = size(X, DIMVARX) + size(Y, DIMVARY) ,
- size(XYSINGVEC, 2) <= min(size(X,DIMVARX) ,size(Y,DIMVARY)).

Further Details

The subroutine computes the basic univariate statistics and the correlation (or covariance) matrix with only one pass through the data.

If fewer than two valid observations were present, the statistics XSTAT, YSTAT, XYSINGVAL XYCOR , XYSINGVAR and XYSINGVEC are set to Nan code.

This subroutine may be used in a call with no observations (e.g., size(X,3-DIMVARX) = 0) in order to finish the computations with LAST=true when the total number of observations is unknown at the beginning of the computations.

6.12.40 subroutine comp_mca_miss (x, y, first, last, xstat, ystat, xycor, xymiss, failure, dimvarx, dimvary, cov, sort, maxiter, xysingval, xysingvar, ysingvec)

Purpose

COMP_MCA_MISS performs Maximum Covariance Analysis (MCA) or canonical covariance analysis between two data matrices XX and YY possibly containing missing values.

COMP_MCA_MISS computes the singular value decomposition (SVD) of the correlation (or covariance) matrix XYCOR between two data matrices XX and YY. This SVD is written

XYCOR = U * SIGMA * V'

where SIGMA is a m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is a m-by-m orthogonal matrix, and V is a n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of XYCOR; they are real and non-negative. The first min(m,n) columns of U and V are the left and right singular vectors of XYCOR.

The routine returns the singular values, the left and, optionally, the right singular vectors of the correlation (or covariance) matrix XYCOR between two data matrices XX and YY.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVARX) observations on size(X,DIMVARX) variables from the "left" matrix of data XX. By default, DIMVARX is equal to 1. See description of optional DIMVARX argument for details. If all the data are available at once, X can be the full data matrix XX.

The shape of X must verify: size(X, 3-DIMVARX) = size(Y, 3-DIMVARY) .

Y (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(Y,3-DIMVARY) observations on size(Y,DIMVARY) variables from the "right" matrix of data YY. By default, DIMVARY is equal to 1. See description of optional DIMVARY argument for details. If all the data are available at once, Y can be the full data matrix YY.

The shape of Y must verify: size(Y, 3-DIMVARY) = size(X, 3-DIMVARX) .

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrices are the first submatrices of the data matrices XX and YY.
- FIRST = false the current submatrices are not the first submatrices of the data matrices XX and YY.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrices are the last submatrices of the data matrices XX and YY.
- LAST = false the current submatrix are not the last submatrices of the data matrices XX and YY.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,4) On entry, after the first call to COMP_MCA_MISS (e.g., when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_MCA_MISS. XSTAT should not be changed between calls to COMP_MCA_MISS.

On exit, when LAST=true, XSTAT contains the following statistics on all variables from the XX matrix:

- XSTAT(:,1) contains the mean values of the "left" data matrix XX.
- XSTAT(:,2) contains the standard-deviations of the "left" data matrix XX.
- XSTAT(:,3) contains the the numbers of non-missing observations in the "left" data matrix XX.
- XSTAT(:,4) is used as workspace.

The shape of XSTAT must verify:

- size(XSTAT, 1) = size(X, DIMVARX) ,
- size(XSTAT, 2) = 4 .

YSTAT (INPUT/OUTPUT) real(stnd), dimension(:,4) On entry, after the first call to COMP_MCA_MISS (e.g., when FIRST=true), YSTAT is used as workspace to accumulate quantities on previous calls to COMP_MCA_MISS. YSTAT should not be changed between calls to COMP_MCA_MISS.

On exit, when LAST=true, YSTAT contains the following statistics on all variables from the YY matrix:

- YSTAT(:,1) contains the mean values of the "right" data matrix YY.
- YSTAT(:,2) contains the the standard-deviations of the "right" data matrix YY.
- YSTAT(:,3) contains the the numbers of non-missing observations in the "right" data matrix YY.
- YSTAT(:,4) is used as workspace.

The shape of YSTAT must verify:

- size(YSTAT, 1) = size(Y, DIMVARY) ,
- size(YSTAT, 2) = 4 .

XYCOR (INPUT/OUTPUT) real(stnd), dimension(:,:,4) On entry, after the first call to COMP_MCA_MISS (e.g., when FIRST=true), XYCOR is used as workspace to accumulate quantities on previous calls to COMP_MCA_MISS. XYCOR should not be changed between calls to COMP_MCA_MISS.

On exit, when LAST=true, XYCOR contains the following statistics:

- XYCOR(i,j,1) contains the correlation coefficients between XX(i,:) and YY(j,:) (XX(:,i) and YY(:,j) if DIMVARX=2 and DIMVARY=2).

- XYCOR(i,j,2) contains the incidence values between XX(i,:) and YY(j,:) (XX(:,i) and YY(:,j) if DIMVARX=2 and DIMVARY=2). XYCOR(i,j,2) indicates the numbers of non-missing pairs of observations which were used in the calculation of XYCOR(i,j,1).
- XYCOR(:,:,3) contains the first min(size(X,DIMVARX),size(Y,DIMVARY)) columns of U, the left singular vectors of the correlation (or covariance) matrix XYCOV between XX and YY.
- XYCOR(:,:,4) is used as workspace.

The shape of XYCOR must verify:

- size(XYCOR, 1) = size(X, DIMVARX) ,
- size(XYCOR, 2) = size(Y, DIMVARY) ,
- size(XYCOR, 3) = 4 .

XYMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X or Y which is equal to XYMISS is assumed to be missing or invalid. The basic univariate statistics and correlation (or covariance) matrix are computed on all the observations where X and Y are not missing (see Further Details).

FAILURE (OUTPUT) logical(lgl) On exit when LAST=true:

- FAILURE = false: indicates successful exit.
- FAILURE = true: indicates that fewer than two valid observations were present for some pair of variables or that the observations on some variable were constant and the correlations were requested or that maximum accuracy was not achieved when computing the SVD of the covariance (or correlation) matrix between the data matrices XX and YY .

On exit when LAST=false, FAILURE is always set to false.

DIMVARX (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVARX is present, DIMVARX is used as follows, if:

- DIMVARX = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVARX = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables, respectively.

The default is DIMVARX = 1.

DIMVARY (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVARY is present, DIMVARY is used as follows, if:

- DIMVARY = 1, the input submatrix Y contains size(Y,2) observations on size(Y,1) variables.
- DIMVARY = 2, the input submatrix Y contains size(Y,1) observations on size(Y,2) variables, respectively.

The default is DIMVARY = 1.

COV (INPUT, OPTIONAL) logical(lgl) On entry, if COV is present and COV=true, a covariance matrix between the data matrices XX and YY is computed instead of a correlation matrix.

By default, a correlation matrix is computed.

COV needs to be specified only on the last call to COMP_MCA_MISS (e.g., when LAST=true).

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The singular vectors are rearranged accordingly.

SORT needs to be specified only on the last call to COMP_MCA (e.g., when LAST=true).

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm. The bidiagonal SVD algorithm of an intermediate bidiagonal form B of XYCOR fails to converge if the number of QR sweeps exceeds MAXITER * min(m,n). Convergence usually occurs in about 2 * min(m,n) QR sweeps.

The default is 10.

MAXITER needs to be specified only on the last call to COMP_MCA (e.g., when LAST=true).

XYSINGVAL (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, XYSINGVAL contains the singular values of the correlation (or covariance) matrix XYCOR between the data matrices XX and YY. If this optional argument is absent, the singular values are stored in XSTAT(:,4), when LAST=true.

The size of XYSINGVAL must verify: size(XYSINGVAL) = min(size(X,DIMVARX), size(Y,DIMVARY)).

YSINGVEC (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, when LAST=true and YSINGVEC is present, YSINGVEC contains the first min(size(X,DIMVARX),size(Y,DIMVARY)) columns of V, the right singular vectors of the correlation (or covariance) matrix XYCOR between XX and YY.

YSINGVEC needs to be specified only on the last call to COMP_MCA_MISS (e.g., when LAST=true).

The shape of YSINGVEC must verify:

- size(YSINGVEC, 1) = size(Y, DIMVARY) ,
- size(YSINGVEC, 2) = min(size(X,DIMVARX), size(Y,DIMVARY)) .

XYSINGVAR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XYSINGVAR is present, XYSINGVAR contains percentages of total squared covariance associated with the left and right singular vectors in order of the singular values stored in XYSINGVAL.

XYSINGVAR needs to be specified only on the last call to COMP_MCA_MISS (e.g., when LAST=true).

The size of XYSINGVAR must verify: size(XYSINGVAR) = min(size(X,DIMVARX), size(Y,DIMVARY)).

Further Details

The subroutine computes the basic univariate statistics and the correlation (or covariance) matrix with only one pass through the data.

If fewer than two valid observations were present for some pair of variables or if the observations on some variable were constants, the statistics XYSINGVAL, XYSINGVEC, XYSINGVAR and XYCOR(:,:,1) are globally set to XMISS.

The means and standard-deviations of XX and YY are computed from all valid data. The correlation coefficients are based on these univariate statistics and on all valid pairs of observations. The singular vectors and singular values are computed from these bivariate statistics.

This subroutine may be used in a call with no observations (e.g., size(X,3-DIMVARX) = 0) in order to finish the computations with LAST=true when the total number of observations is unknown at the beginning of the computations.

6.12.41 subroutine comp_mca_miss2 (x, y, first, last, xstat, ystat, xycor, xymiss, failure, dimvarx, dimvary, cov, xysingval, maxiter, ortho, xysingvar, xysingvec)

Purpose

COMP_MCA_MISS2 performs Maximum Covariance Analysis (MCA) or canonical covariance analysis between two data matrices XX and YY possibly containing missing values.

COMP_MCA_MISS2 computes a partial singular value decomposition (SVD) of the correlation (or covariance) matrix XYCOR between two data matrices XX and YY. This partial SVD is written

U(:m,:k) * SIGMA(:k,:k) * V(:n,:k)'

where SIGMA is a k-by-k matrix which is zero except for its k diagonal elements, U is a m-by-k orthogonal matrix, and V is a n-by-k orthogonal matrix. The diagonal elements of SIGMA are the first k singular values of XYCOR; they are real and non-negative. The k columns of U and V are the first k left and right singular vectors of XYCOR.

COMP_MCA_MISS2 computes all the singular values, and, optionally, selected left and right singular vectors (by inverse iteration), of the covariance (or correlation matrix) XYCOR between two data matrices XX and YY.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVARX) observations on size(X,DIMVARX) variables from the "left" matrix of data XX. By default, DIMVARX is equal to 1. See description of optional DIMVARX argument for details. If all the data are available at once, X can be the full data matrix XX.

The shape of X must verify: size(X, 3-DIMVARX) = size(Y, 3-DIMVARY) .

Y (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(Y,3-DIMVARY) observations on size(Y,DIMVARY) variables from the "right" matrix of data YY. By default, DIMVARY is equal to 1. See description of optional DIMVARY argument for details. If all the data are available at once, Y can be the full data matrix YY.

The shape of Y must verify: size(Y, 3-DIMVARY) = size(X, 3-DIMVARX) .

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrices are the first submatrices of the data matrices XX and YY.
- FIRST = false the current submatrices are not the first submatrices of the data matrices XX and YY.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrices are the last submatrices of the data matrices XX and YY.
- LAST = false the current submatrix are not the last submatrices of the data matrices XX and YY.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,4) On entry, after the first call to COMP_MCA_MISS2 (e.g., when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_MCA_MISS2. XSTAT should not be changed between calls to COMP_MCA_MISS2.

On exit, when LAST=true, XSTAT contains the following statistics on all variables from the XX matrix:

- XSTAT(:,1) contains the mean values of the "left" data matrix XX.
- XSTAT(:,2) contains the standard-deviations of the "left" data matrix XX.
- XSTAT(:,3) contains the the numbers of non-missing observations in the "left" data matrix XX.
- XSTAT(:,4) is used as workspace.

The shape of XSTAT must verify:

- size(XSTAT, 1) = size(X, DIMVARX) ,
- size(XSTAT, 2) = 4 .

YSTAT (INPUT/OUTPUT) real(stnd), dimension(:,4) On entry, after the first call to COMP_MCA_MISS2 (e.g., when FIRST=true), YSTAT is used as workspace to accumulate quantities on previous calls to COMP_MCA_MISS2. YSTAT should not be changed between calls to COMP_MCA_MISS2.

On exit, when LAST=true, YSTAT contains the following statistics on all variables from the YY matrix:

- YSTAT(:,1) contains the mean values of the "right" data matrix YY.
- YSTAT(:,2) contains the the standard-deviations of the "right" data matrix YY.
- YSTAT(:,3) contains the the numbers of non-missing observations in the "right" data matrix YY.
- YSTAT(:,4) is used as workspace.

The shape of YSTAT must verify:

- size(YSTAT, 1) = size(Y, DIMVARY) ,
- size(YSTAT, 2) = 4 .

XYCOR (INPUT/OUTPUT) real(stnd), dimension(:,:,4) On entry, after the first call to COMP_MCA_MISS2 (e.g., when FIRST=true), XYCOR is used as workspace to accumulate quantities on previous calls to COMP_MCA_MISS2. XYCOR should not be changed between calls to COMP_MCA_MISS2.

On exit, when LAST=true, XYCOR contains the following statistics:

- XYCOR(i,j,1) contains the correlation coefficients between XX(i,:) and YY(j,:) (XX(:,i) and YY(:,j) if DIMVARX=2 and DIMVARY=2).
- XYCOR(i,j,2) contains the incidence values between XX(i,:) and YY(j,:) (XX(:,i) and YY(:,j) if DIMVARX=2 and DIMVARY=2). XYCOR(i,j,2) indicates the numbers of non-missing pairs of observations which were used in the calculation of XYCOR(i,j,1).
- XYCOR(:,:,3:4) is used as workspace.

The shape of XYCOR must verify:

- size(XYCOR, 1) = size(X, DIMVARX) ,
- size(XYCOR, 2) = size(Y, DIMVARY) ,
- size(XYCOR, 3) = 4 .

XYMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X or Y which is equal to XYMISS is assumed to be missing or invalid. The basic univariate statistics and correlation (or covariance) matrix are computed on all the observations where X and Y are not missing (see Further Details).

FAILURE (OUTPUT) logical(lgl) On exit when LAST=true:

- FAILURE = false: indicates successful exit.
- FAILURE = true: indicates that fewer than two valid observations were present for some pair of variables or that the observations on some variable were constant and the correlations were requested or that maximum accuracy was not achieved when computing the eigenvalues or that some eigenvectors failed to converge with MAXITER inverse iterations.

On exit when LAST=false, FAILURE is always set to false.

DIMVARX (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVARX is present, DIMVARX is used as follows, if:

- DIMVARX = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVARX = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables, respectively.

The default is DIMVARX = 1.

DIMVARY (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVARY is present, DIMVARY is used as follows, if:

- DIMVARY = 1, the input submatrix Y contains size(Y,2) observations on size(Y,1) variables.
- DIMVARY = 2, the input submatrix Y contains size(Y,1) observations on size(Y,2) variables, respectively.

The default is DIMVARY = 1.

COV (INPUT, OPTIONAL) logical(lgl) On entry, if COV is present and COV=true, a covariance matrix between the data matrices XX and YY is computed instead of a correlation matrix.

By default, a correlation matrix is computed.

COV needs to be specified only on the last call to COMP_MCA_MISS2 (e.g., when LAST=true).

XYSINGVAL (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, XYSINGVAL contains the singular values of the correlation (or covariance) matrix XYCOR between the data matrices XX and YY. If this optional argument is absent, the singular values are stored in XSTAT(:,4), when LAST=true.

The size of XYSINGVAL must verify: size(XYSINGVAL) = min(size(X,DIMVARX), size(Y,DIMVARY)) .

MAXITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed in the subroutine for computing the singular vectors. By default, 2 inverse iterations are performed for all the singular vectors. This optional argument is used only if the XYSINGVEC argument is present.

MAXITER needs to be specified only on the last call to COMP_MCA2 (e.g., when LAST=true).

ORTHO (INPUT, OPTIONAL) logical(lgl) If ORTHO=true the computed singular vectors are orthogonalized by the Modified Gram-Schmidt algorithm. This optional argument is used only if the XYSINGVEC argument is present.

The default is FALSE.

ORTHO needs to be specified only on the last call to COMP_MCA2 (e.g., when LAST=true).

XYSINGVAR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XYSINGVAR is present, XYSINGVAR contains percentages of total squared covariance associated with the left and right singular vectors in order of the singular values stored in XYSINGVAL.

XYSINGVAR needs to be specified only on the last call to COMP_MCA_MISS2 (e.g., when LAST=true).

The size of XYSINGVAR must verify: size(XYSINGVAR) = min(size(X,DIMVARX), size(Y,DIMVARY)).

XYSINGVEC (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, when LAST=true and XYSINGVEC is present, XYSINGVEC contains the first columns of U and V, the first k left and right singular vectors of the correlation (or covariance) matrix XYCOR between XX and YY.

The first k left singular vectors are stored in XYSINGVEC(:size(X,DIMVARX),:) The first k right singular vectors are stored in XYSINGVEC(size(X,DIMVARX)+1:,:)

XYSINGVEC needs to be specified only on the last call to COMP_MCA2 (e.g., when LAST=true).

The shape of XYSINGVEC must verify:

- size(XYSINGVEC, 1) = size(X, DIMVARX) + size(Y, DIMVARY) ,
- size(XYSINGVEC, 2) <= min(size(X,DIMVARX) ,size(Y,DIMVARY)) .

Further Details

The subroutine computes the basic univariate statistics and the correlation (or covariance) matrix with only one pass through the data.

If fewer than two valid observations were present for some pair of variables or if the observations on some variable were constants, the statistics XYSINGVAL, XYSINGVEC, XYSINGVAR and XYCOR(:,:,1) are globally set to XMISS.

The means and standard-deviations of XX and YY are computed from all valid data. The correlation coefficients are based on these univariate statistics and on all valid pairs of observations. The singular vectors and singular values are computed from these bivariate statistics.

This subroutine may be used in a call with no observations (e.g., size(X,3-DIMVARX) = 0) in order to finish the computations with LAST=true when the total number of observations is unknown at the beginning of the computations.

6.12.42 subroutine comp_pc_mca (x, xsingvec, first, last, xpccor, pccorp, xpc, xn, dimvar, xmean, xstd, xpcvar)

Purpose

COMP_PC_MCA computes estimates of Singular Variables (SV) and correlation (or covariance) fields from a data matrix XX and a set of singular vectors derived from MCA analysis of XX with another matrix YY.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which Singular Variables are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix.

XSINGVEC (INPUT) real(stnd), dimension(:,:) On entry, XSINGVEC contains selected left (or right) singular vectors of the SVD of the covariance (or correlation) matrix between the data matrix XX and another data matrix YY.

The shape of XSINGVEC must verify: size(XSINGVEC, 1) = size(X, DIMVAR).

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrix is the first submatrix of the data matrix.
- FIRST = false the current submatrix is not the first submatrix of the data matrix.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrix is the last submatrix of the data matrix.
- LAST = false the current submatrix is not the last submatrix of the data matrix.

XPCCOR (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_PC_MCA (e.g., when FIRST=true), XPCCOR is used as workspace to accumulate quantities on previous calls to COMP_PC_MCA. XPCCOR should not be changed between calls to COMP_PC_MCA.

On exit, when LAST=true, XPCCOR contains:

- the correlations between the data matrix and the Singular Variables if the optional arguments XMEAN and XSTD are present.
- the covariances between the data matrix and the normalized Singular Variables if only the optional argument XMEAN is present.

The shape of XPCCOR must verify:

- size(XPCCOR, 1) = size(X, DIMVAR) ,
- size(XPCCOR, 2) = size(XSINGVEC, 2) .

PCCORP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_PC_MCA (e.g., when FIRST=true), PCCORP is used as workspace to accumulate quantities on previous calls to COMP_PC_MCA. PCCORP should not be changed between calls to COMP_PC_MCA.

On exit, when LAST=true, PCCORP contains the correlation matrix between the Singular Variables stored in argument XPC. PCCORP is stored in symmetric storage mode (see further details).

The size of PCCORP must verify: size(PCCORP) = (size(XSINGVEC,2) * (size(XSINGVEC,2)+1))/2.

XPC (OUTPUT) real(stnd), dimension(:,:) On exit, XPC contains the unnormalized Singular Variables derived from X and XSINGVEC.

The shape of XPC must verify:

- size(XPC, 1) = size(X, 3-DIMVAR) ,
- size(XPC, 2) = size(XSINGVEC, 2) .

XN (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_PC_MCA (e.g., when FIRST=true), XN contains count of observations from previous calls to COMP_PC_MCA. XN should not be changed between calls to COMP_PC_MCA.

On exit, XN contains the number of observations in the data matrix XX.

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows, if:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

XMEAN (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if XMEAN is present, XMEAN contains the variable means and the Singular Variables are computed from the centered data matrix X.

The size of XMEAN must verify: size(XMEAN) = size(X, DIMVAR).

XSTD (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if XSTD is present, XSTD contains the variable standard-deviations and the Singular Variables are computed from the normalized data matrix X.

The size of XSTD must verify: size(XSTD) = size(X, DIMVAR).

XPCVAR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, when LAST=true and XPCVAR is present, XPCVAR contains the variances of the Singular Variables stored in argument XPC.

The size of XPCVAR must verify: size(XPCVAR) = size(XSINGVEC, 2)

XPCVAR needs to be specified only on the last call to COMP_PC_MCA (e.g., when LAST=true).

Further Details

The subroutine computes the Singular Variables and the correlations matrices with only one pass through the data.

On exit, the upper triangle of the symmetric correlation matrix COR between the Singular Variables is packed columnwise in the linear array PCCORP. More precisely, the j-th column of this matrix COR is stored in the array PCCORP as follows:

PCCORP(i + (j-1) * j/2) = COR(i,j) for 1<=i<=j;

If fewer than two valid observations were present, the arguments XPCVAR, XPCCOR and PCCORP are set to nan() code.

This subroutine may be used in a call with no observations (e.g., size(X,3-DIMVAR) = size(XPC,1) = 0) in order to finish the computations with LAST=true when the total number of observations is unknown at the beginning of the computations.

The correlations (or covariance) between the Singular variables XPC and the data array YY may be computed easily from the outputs of COMP_MCA and COMP_PC_MCA. If YSINGVEC(:p,:k) are the k singular vectors (derived from data array YY) associated with the k singular values XYSINGVAL(:k) and the k singular vectors XSINGVEC(:m,:k) (derived from data array XX). Then, the correlations (or covariance) between the Singular variables XPC and the data array YY are equal to

spread(XYSINGVAL(:k)/SQRT(XPCVAR(:k)), dim=1, ncopies=p) * YSINGVEC(:p,:k)

This matrix is a covariance matrix between data array YY and the normalized Singular Variables XPC if a covariance matrix between data arrays XX and YY is analysed or a correlation matrix between data array YY and the Singular Variables XPC if a correlation matrix between data arrays XX and YY is analysed.

6.12.43 `subroutine comp_pc ( x, xeigvec, xpc, dimvar, xmean, xstd, xsingval )`

Purpose

COMP_PC computes estimates of a Principal Component (PC) from a data matrix XX and an eigenvector or singular vector derived from EOF or MCA analysis.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data XX for which the Principal Component is desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix XX.

XEIGVEC (INPUT) real(stnd), dimension(:) On entry, XEIGVEC contains an eigenvector of the variance-covariance (or correlation) matrix from the data matrix XX or a selected left (or right) singular vector of the SVD of the covariance matrix between the data matrix XX and another data matrix YY.

The shape of XEIGVEC must verify: size(XEIGVEC) = size(X, DIMVAR).

XPC (OUTPUT) real(stnd), dimension(:) On exit, XPC contains the Principal Component derived from X and XEIGVEC.

The size of XPC must verify: size(XPC) = size(X, 3-DIMVAR).

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows, if:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

XMEAN (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if XMEAN is present, XMEAN contains the variable means and the Principal Component is computed from the centered data matrix X.

The size of XMEAN must verify: size(XMEAN) = size(X, DIMVAR).

XSTD (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if XSTD is present, XSTD contains the variable standard-deviations and the Principal Component is computed from the normalized data matrix X.

The size of XSTD must verify: size(XSTD) = size(X, DIMVAR).

XSINGVAL (INPUT, OPTIONAL) real(stnd) On entry, XSINGVAL must contain the singular value of the covariance (or correlation) matrix from the data matrix XX associated with the eigenvector in XEIGVEC array. If SINGVAL is present, the Principal Component is normalized by XSINGVAL on output (the variance of the Principal Component is equal to one). This optional argument is useful only if XEIGVEC contains an eigenvector derived from an EOF analysis.

Further Details

The subroutine computes the Principal Component with only one pass through the data.

6.12.44 `subroutine comp_pc ( x, xeigvec, xpc, dimvar, xmean, xstd, xsingval )`

Purpose

COMP_PC computes estimates of Principal Components (PC) from a data matrix XX and a set of eigenvectors or singular vectors derived from EOF or MCA analysis.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data XX for which Principal Components are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix XX.

XEIGVEC (INPUT) real(stnd), dimension(:,:) On entry, XEIGVEC contains selected eigenvectors of the variance-covariance (or correlation) matrix from the data matrix XX or selected left (or right) singular vectors of the SVD of the covariance matrix between the data matrix XX and another data matrix YY.

The shape of XEIGVEC must verify: size(XEIGVEC, 1) = size(X, DIMVAR).

XPC (OUTPUT) real(stnd), dimension(:,:) On exit, XPC contains the Principal Components derived from X and XEIGVEC.

The shape of XPC must verify:

- size(XPC, 1) = size(X, 3-DIMVAR) ,
- size(XPC, 2) = size(XEIGVEC, 2) .

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows, if:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

XMEAN (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if XMEAN is present, XMEAN contains the variable means and the Principal Components are computed from the centered data matrix X.

The size of XMEAN must verify: size(XMEAN) = size(X, DIMVAR).

XSTD (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if XSTD is present, XSTD contains the variable standard-deviations and the Principal Components are computed from the normalized data matrix X.

The size of XSTD must verify: size(XSTD) = size(X, DIMVAR).

XSINGVAL (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, XSINGVAL must contain the singular values of the covariance (or correlation) matrix from the data matrix XX associated with the eigenvectors in XEIGVEC array. If SINGVAL is present, the Principal Components are normalized by XSINGVAL on output (the variances of the Principal Components are equal to one). This optional argument is useful only if XEIGVEC contains eigenvectors derived from an EOF analysis.

The size of XSINGVAL must verify: size(XSINGVAL) = size(XEIGVEC, 2).

Further Details

The subroutine computes the Principal Components with only one pass through the data.

6.12.45 `subroutine comp_pc_miss ( x, xeigvec, xpc, xmiss, dimvar, xmean, xstd, xsingval )`

Purpose

COMP_PC_MISS computes estimates of a Principal Component (PC) from a data matrix XX and an eigenvector or singular vector derived from EOF or MCA analysis when XX contains missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data XX for which a Principal Components is desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix XX.

XEIGVEC (INPUT) real(stnd), dimension(:) On entry, XEIGVEC contains a selected eigenvector of the variance-covariance (or correlation) matrix from the data matrix XX or a selected left (or right) singular vector of the SVD of the covariance matrix between the data matrix XX and another data matrix YY.

The shape of XEIGVEC must verify: size(XEIGVEC) = size(X, DIMVAR).

XPC (OUTPUT) real(stnd), dimension(:) On exit, XPC contains the Principal Component derived from X and XEIGVEC.

The shape of XPC must verify: size(XPC) = size(X, 3-DIMVAR).

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing or invalid. The Principal Components are computed from all the observations where X X are not missing by regression (see Further Details).

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows, if:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

XMEAN (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if XMEAN is present, XMEAN contains the variable means and the Principal Component is computed from the centered data matrix X.

The size of XMEAN must verify: size(XMEAN) = size(X, DIMVAR).

XSTD (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if XSTD is present, XSTD contains the variable standard-deviations and the Principal Component is computed from the normalized data matrix X.

The size of XSTD must verify: size(XSTD) = size(X, DIMVAR).

XSINGVAL (INPUT, OPTIONAL) real(stnd) On entry, XSINGVAL must contain the singular value of the covariance (or correlation) matrix from the data matrix XX associated with the eigenvector in XEIGVEC array. If SINGVAL is present, the Principal Component is normalized by XSINGVAL on output. This optional argument is useful only if XEIGVEC contains an eigenvector derived from an EOF analysis.

Further Details

The subroutine computes the Principal Component with only one pass through the data by regressing the observations onto the eigenvector of the correlation or covariance matrix of X.

If for some observations in X there is no available data, the corresponding element in XPC is filled with XMISS.

6.12.46 subroutine comp_pc_miss (x, xeigvec, xpc, xmiss, dimvar, xmean, xstd, xsingval, tol, min_norm)

Purpose

COMP_PC_MISS computes estimates of Principal Components (PC) from a data matrix XX and a set of eigenvectors or singular vectors derived from EOF or MCA analysis when XX contains missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data XX for which Principal Components are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix XX.

XEIGVEC (INPUT) real(stnd), dimension(:,:) On entry, XEIGVEC contains selected eigenvectors of the variance-covariance (or correlation) matrix from the data matrix XX or selected left (or right) singular vectors of the SVD of the covariance matrix between the data matrix XX and another data matrix YY.

The shape of XEIGVEC must verify: size(XEIGVEC, 1) = size(X, DIMVAR).

XPC (OUTPUT) real(stnd), dimension(:,:) On exit, XPC contains the Principal Components derived from X and XEIGVEC.

The shape of XPC must verify:

- size(XPC, 1) = size(X, 3-DIMVAR) ,
- size(XPC, 2) = size(XEIGVEC, 2) .

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing or invalid. The Principal Components are computed from all the observations where X X are not missing by regression (see Further Details).

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows, if:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

XMEAN (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if XMEAN is present, XMEAN contains the variable means and the Principal Components are computed from the centered data matrix X.

The size of XMEAN must verify: size(XMEAN) = size(X, DIMVAR).

XSTD (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if XSTD is present, XSTD contains the variable standard-deviations and the Principal Components are computed from the normalized data matrix X.

The size of XSTD must verify: size(XSTD) = size(X, DIMVAR).

XSINGVAL (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, XSINGVAL must contain the singular values of the covariance (or correlation) matrix from the data matrix XX associated with the eigenvectors in XEIGVEC array. If SINGVAL is present, the Principal Components are normalized by XSINGVAL on output. This optional argument is useful only if XEIGVEC contains eigenvectors derived from an EOF analysis.

The size of XSINGVAL must verify: size(XSINGVAL) = size(XEIGVEC, 2).

TOL (INPUT, OPTIONAL) real(stnd) On entry, TOL is used to determine the effective rank of the coefficient matrix A for each regression problem, which is then defined as the order of the largest leading triangular submatrix R11 in the QR factorization (with pivoting) of A whose estimated condition number, in the 1-norm, is less than 1/TOL. TOl must be set to the relative precision of the elements in A and B. If each element is correct to, say, 5 digits then TOL=0.00001 should be used.

TOL must not be greater or equal to 1 or less or equal than 0, otherwise the numerical rank of A is determined and the calculations to determine the condition number are not performed. If TOL is absent, the numerical rank of A is determined for each regression problem.

MIN_NORM (INPUT, OPTIONAL) logical(lgl) On entry, If MIN_NORM=true, minimun 2-norm solutions are computed. If MIN_NORM=false or if MIN_NORM is absent, solutions are computed such that if the j-th column of XEIGVEC is omitted from the basis for the ith observation (regression problem), X[i,j] is set to zero.

Further Details

The subroutine computes the Principal Components with only one pass through the data by regressing the observations onto the eigenvectors of the correlation or covariance matrix of X.

If for some observations in X there is no available data, the corresponding line in XPC is filled with XMISS.

6.13 Module_Num_Constants

Copyright 2024 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

THIS MODULE PROVIDES SIMPLE NAMES AND ROUTINES FOR THE VARIOUS MACHINE DEPENDENT CONSTANTS. ALL ARE FOR PRECISION ‘stnd’.

LATEST REVISION : 25/02/2024

6.13.1 `function lamch ( cmach )`

Purpose

LAMCH determines machine parameters for precision STND.

Arguments

CMACH (INPUT) character*1 Specifies the value to be returned by LAMCH. If:

- CMACH = 'S' or 's', LAMCH := sfmin
- CMACH = 'T' or 't', LAMCH := t
- CMACH = 'R' or 'r', LAMCH := rnd
- CMACH = 'G' or 'g', LAMCH := grd
- CMACH = 'U' or 'u', LAMCH := unitrnd
- CMACH = 'P' or 'p', LAMCH := prec

where:

- sfmin = safe minimum, such that 1/sfmin does not overflow.
- t = number of (base) digits in the floating-point significand.
- **rnd = 0.0 when floating-point addition rounds upward, downward**

  or toward zero;

  = 1.0 when floating-point addition rounds to nearest, but not in the IEEE style;

  = 2.0 when floating-point addition rounds in the IEEE style.
- **grd = 1. if floating-point arithmetic chops (rnd = 0.) and more**

  than t digits participate in the post-normalization shift of the floating-point significand in multiplication,

  = 0.0 otherwise.
- **unitrnd = unit roundoff of the machine, e.g., the maximum** relative representation error of a real number in the range of the floating point numbers of kind STND.
- **prec = unitrnd*machbase, e.g., the relative spacing between** consecutive floating point numbers in the range of the floating point numbers of kind STND.

Further Details

The routine is based on the routine DLAMCH in LAPACK77 (version 3). Note that the interface of DLAMCH in more recent versions of LAPACK has changed and is now using intrinsic Fortran90 functions to get the values of the machine parameters.

For any other characters, LAMCH returns the bit pattern corresponding to a quiet NaN.

6.13.2 subroutine mach (basedigits, irnd, iuflow, igrd, iexp, ifloat, expepspos, expepsneg, minexpbase, maxexpbase, epspos, epsneg, epsilpos, epsilneg, rndunit)

Purpose

This subroutine is intended to determine the parameters of the floating-point arithmetic system specified below.

Arguments

BASEDIGITS (OUTPUT, OPTIONAL) integer(i4b) The number of base digits in the floating-point significand.

IRND (OUTPUT, OPTIONAL) integer(i4b) A parameter indicating whether proper rounding or chopping (rounding upward, downward, toward zero) occurs in addition. If:

- IRND = 0 if floating-point addition rounds upward, downward or toward zero;
- IRND = 1 if floating-point addition rounds to nearest, but not in the IEEE style;
- IRND = 2 if floating-point addition rounds in the IEEE style.

IUFLOW (OUTPUT, OPTIONAL) integer(i4b) A parameter indicating whether underflow is full or partial:

- IUFLOW = 0 if there is full underflow (flush to zero, etc);
- IUFLOW = 1 if there is partial underflow.

IGRD (OUTPUT, OPTIONAL) integer(i4b) The number of guard digits for multiplication with chopping arithmetic (IRND = 0). If:

- IGRD = 0 if floating-point arithmetic rounds, or if it chops and only BASEDIGITS digits participate in the post-normalization shift of the floating-point significand in multiplication;
- IGRD = 1 if floating-point arithmetic chops and more than BASEDIGITS digits participate in the post-normalization shift of the floating-point significand in multiplication.

IEXP (OUTPUT, OPTIONAL) integer(i4b) A guess for the number of bits dedicated to the representation of the exponent of a floating point number if BASE is a power of two and -1 otherwise.

IFLOAT (OUTPUT, OPTIONAL) integer(i4b) A guess for the number of bits dedicated to the representation of a floating point number if BASE is a power of two and -1 otherwise.

EXPEPSPOS (OUTPUT, OPTIONAL) integer(i4b) The largest in magnitude negative integer such that

```
1.0 + float(base)**(expepspos) /= 1.
```

EXPEPSNEG (OUTPUT, OPTIONAL) integer(i4b) The largest in magnitude negative integer such that

```
1.0 - float(base)**(expepsneg) /= 1.
```

MINEXPBASE (OUTPUT, OPTIONAL) integer(i4b) The largest in magnitude negative integer such that float(base)**minexpbase is positive and normalized.

MAXEXPBASE (OUTPUT, OPTIONAL) integer(i4b) The largest in magnitude positive integer such that float(base)**(maxexpbase) is positive and normalized.

EPSPOS (OUTPUT, OPTIONAL) real(stnd) The smallest power of BASE whose sum with 1. is greater than 1. That is, float(base)**(expepspos).

EPSNEG (OUTPUT, OPTIONAL) real(stnd) The smallest power of BASE whose difference with 1. is less than 1. That is, float(base)**(expepsneg).

EPSILPOS (OUTPUT, OPTIONAL) real(stnd) The smallest positive floating point number whose sum with 1. is greater than 1.

EPSILNEG (OUTPUT, OPTIONAL) real(stnd) The smallest positive floating point number whose difference with 1. is less than 1.

RNDUNIT (OUTPUT, OPTIONAL) real(stnd) unit roundoff of the machine, e.g., machine epsilon of the machine.

Further Details

This subroutine is based on the routines MACHAR by Cody and DLAMCH in LAPACK77 (version 3). For further details, See:

(1) **Malcolm M.A., 1972:** Algorithms to reveal properties of floating-point arithmetic. Comms. of the ACM, 15, 949-951.

(2) **Gentleman, W.M., and Marovich, S.B., 1974:** More on algorithms that reveal properties of floating point arithmetic units. Comms. of the ACM, 17, 276-277.

(3) **Cody, W.J., 1988:** MACHAR: A subroutine to dynamically determine machine parameters. TOMS 14, No. 4, 303-311.

6.13.3 `function test_ieee ( )`

Purpose

TEST_IEEE try to determine if the computer follows the IEEE standard 754 for binary floating-point arithmetic.

Arguments

none

Further Details

If the compiler follows the Fortran 2003 standard, the facilities provided by the IEEE_ARITHMETIC module are used to determine if the computer follows the IEEE standard 754 for binary floating-point arithmetic.

6.13.4 `function test_nan ( )`

Purpose

TEST_NAN returns TRUE if NaNs exist, and FALSE otherwise.

Arguments

None

Further Details

If the compiler follows the Fortran 2003 standard, the facilities provided by the IEEE_ARITHMETIC module are used to determine if NaNs exist as defined in the IEEE standard 754 for binary floating-point arithmetic.

Otherwise, the routine exploits the IEEE requirement that NaNs compare as unequal to all values, including themselves.

For further details, see:

(1) **Cody, W.J., and Coonen, J.T., 1993:** Algorithm 722, TOMS 19, No. 4, 443-451.

6.13.5 `function is_nan ( x )`

Purpose

This function returns TRUE if the scalar X is a NaN, and FALSE otherwise.

Arguments

X (INPUT) real(stnd) The floating point number to be tested.

Further Details

If the compiler follows the Fortran 2003 standard, the facilities provided by the IEEE_ARITHMETIC module are used to detect NaNs as defined in the IEEE standard 754 for binary floating-point arithmetic.

If the IEEE_ARITHMETIC module is not available, but the compiler supports the intrinsic function isnan(), this function is used to detect NaNs.

Otherwise, the routine exploits the IEEE requirement that NaNs compare as unequal to all values, including themselves.

Finally, if the computer does not follow the IEEE standard 754 for binary floating-point arithmetic, this function returns TRUE if the scalar X is equal to huge(X).

For further details, see:

(1) **Cody, W.J., and Coonen, J.T., 1993:** Algorithm 722, TOMS 19, No. 4, 443-451.

6.13.6 `function is_nan ( x )`

Purpose

This function returns the value TRUE if any of the elements of the vector X is a NaN, and FALSE otherwise.

Arguments

X (INPUT) real(stnd), dimension(:) The floating point vector to be tested.

Further Details

If the compiler follows the Fortran 2003 standard, the facilities provided by the IEEE_ARITHMETIC module are used to detect NaNs as defined in the IEEE standard 754 for binary floating-point arithmetic.

If the IEEE_ARITHMETIC module is not available, but the compiler supports the intrinsic function isnan(), this function is used to detect NaNs.

Otherwise, the routine exploits the IEEE requirement that NaNs compare as unequal to all values, including themselves.

If the computer does not follow the IEEE standard 754 for binary floating-point arithmetic, this function returns TRUE if any of the elements of the vector X is equal to huge(X).

For further details, see:

(1) **Cody, W.J., and Coonen, J.T., 1993:** Algorithm 722, TOMS 19, No. 4, 443-451.

6.13.7 `function is_nan ( x )`

Purpose

This function returns the value TRUE if any of the elements of the matrix X is a NaN, and FALSE otherwise.

Arguments

X (INPUT) real(stnd), dimension(:,:) The floating point matrix to be tested.

Further Details

If the compiler follows the Fortran 2003 standard, the facilities provided by the IEEE_ARITHMETIC module are used to detect NaNs as defined in the IEEE standard 754 for binary floating-point arithmetic.

If the IEEE_ARITHMETIC module is not available, but the compiler supports the intrinsic function isnan(), this function is used to detect NaNs.

Otherwise, the routine exploits the IEEE requirement that NaNs compare as unequal to all values, including themselves.

If the computer does not follow the IEEE standard 754 for binary floating-point arithmetic, this function returns TRUE if any of the elements of the matrix X is equal to huge(X).

For further details, see:

(1) **Cody, W.J., and Coonen, J.T., 1993:** Algorithm 722, TOMS 19, No. 4, 443-451.

6.13.8 subroutine replace_nan (x, missing)

Purpose

This subroutine replaces the scalar X with the scalar MISSING, if X is a NaN on input.

Arguments

X (INPUT/OUTPUT) real(stnd) The floating point number to be tested.

MISSING (INPUT) real(stnd) The floating point number used to replace NaNs.

Further Details

If the compiler follows the Fortran 2003 standard, the facilities provided by the IEEE_ARITHMETIC module are used to detect NaNs as defined in the IEEE standard 754 for binary floating-point arithmetic.

If the IEEE_ARITHMETIC module is not available, but the compiler supports the intrinsic function isnan(), this function is used to detect NaNs.

Otherwise, the routine exploits the IEEE requirement that NaNs compare as unequal to all values, including themselves.

If the computer does not follow the IEEE standard 754 for binary floating-point arithmetic, this subroutine replaces X with the scalar MISSING, if X is equal to huge(X).

For further details, see:

(1) **Cody, W.J., and Coonen, J.T., 1993:** Algorithm 722, TOMS 19, No. 4, 443-451.

6.13.9 subroutine replace_nan (x, missing)

Purpose

This subroutine replaces the elements of the vector X which are NaNs by the scalar MISSING.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:) The floating point vector to be tested.

MISSING (INPUT) real(stnd) The floating point number used to replace the NaNs.

Further Details

If the compiler follows the Fortran 2003 standard, the facilities provided by the IEEE_ARITHMETIC module are used to detect NaNs as defined in the IEEE standard 754 for binary floating-point arithmetic.

If the IEEE_ARITHMETIC module is not available, but the compiler supports the intrinsic function isnan(), this function is used to detect NaNs.

Otherwise, the routine exploits the IEEE requirement that NaNs compare as unequal to all values, including themselves.

Finally, if the computer does not follow the IEEE standard 754 for binary floating-point arithmetic, this subroutine replaces the elements of the vector X which are equal to huge(X) with the scalar MISSING.

For further details, see:

(1) **Cody, W.J., and Coonen, J.T., 1993:** Algorithm 722, TOMS 19, No. 4, 443-451.

6.13.10 `subroutine replace_nan ( x, missing )`

Purpose

This subroutine replaces the elements of the matrix X which are NaNs by the scalar MISSING.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:,:) The floating point matrix to be tested.

MISSING (INPUT) real(stnd) The floating point number used to replace the NaNs.

Further Details

If the compiler follows the Fortran 2003 standard, the facilities provided by the IEEE_ARITHMETIC module are used to detect NaNs as defined in the IEEE standard 754 for binary floating-point arithmetic.

If the IEEE_ARITHMETIC module is not available, but the compiler supports the intrinsic function isnan(), this function is used to detect NaNs.

Otherwise, the routine exploits the IEEE requirement that NaNs compare as unequal to all values, including themselves.

Finally, if the computer does not follow the IEEE standard 754 for binary floating-point arithmetic, this subroutine replaces the elements of the matrix X which are equal to huge(X) with the scalar MISSING.

For further details, see:

(1) **Cody, W.J., and Coonen, J.T., 1993:** Algorithm 722, TOMS 19, No. 4, 443-451.

6.13.11 `function nan ( )`

Purpose

NAN returns as a scalar function, the bit pattern corresponding to a quiet NaN in the IEEE standard 754 for binary floating-point arithmetic if the machine recognizes NaNs or the maximum floating point number of kind STND otherwise.

Arguments

None

Further Details

If the compiler follows the Fortran 2003 standard, the facilities provided by the IEEE_ARITHMETIC module are used to create a quiet NaN as defined in the IEEE standard 754 for binary floating-point arithmetic.

Otherwise, the routine exploits the IEEE requirement that NaNs compare as unequal to all values, including themselves.

Finally, NAN returns the maximum floating point number of kind STND, if the computer does not follow the IEEE standard 754 for binary floating-point arithmetic.

6.13.12 `function true_nan ( )`

Purpose

TRUE_NAN returns as a scalar function, the bit pattern corresponding to a quiet NaN in the IEEE standard 754 for binary floating-point arithmetic.

Arguments

None

6.14 Module_Print_Procedures

Copyright 2022 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING PRINTING UTILITIES.

LATEST REVISION : 27/03/2022

6.14.1 `subroutine enter_proc ( string, level, prt_unit )`

Purpose

Upon entering a procedure, this subroutine will be called. It would skips two lines and outputs a message that the routine identified by STRING was entered. If LEVEL is present, the message is prepended by LEVEL blanks.

Arguments

STRING (INPUT) character(len=*) The string which identifies the routine.

LEVEL (INPUT, OPTIONAL) integer(i4b) The number of blanks to use for indentation.

PRT_UNIT (INPUT, OPTIONAL) integer(i4b) The printing unit.

Further Details

Leading and trailing blanks in STRING are removed. If PRT_UNIT is absent, then all output is on the unit DEFUNIT.

6.14.2 `subroutine leave_proc ( string, level, prt_unit )`

Purpose

This is the 'opposite' to ENTER_PROC. It should be called just before leaving a routine. An exit message is output on PRT_UNIT and two lines are skipped.

Arguments

STRING (INPUT) character(len=*) The string which identifies the routine.

LEVEL (INPUT, OPTIONAL) integer(i4b) The number of blanks to use for indentation.

PRT_UNIT (INPUT, OPTIONAL) integer(i4b) The printing unit.

Further Details

Leading and trailing blanks in STRING are removed. If PRT_UNIT is absent, then all output is on the unit DEFUNIT.

6.14.3 `function entering ( string, level, trace, prt_unit )`

Purpose

Upon entering a procedure, this function will be called. It will return a prefix string suitable for indenting output lines from the procedure. It takes the given STRING and prepends LEVEL blanks, followed by a '[', and appends the character ']'. For example, if STRING were 'hi' and LEVEL were 7, it would return " [hi]". LEVEL is then also incremented by 2.

If TRACE is true, it also outputs a message that the routine identified by STRING was entered.

Arguments

STRING (INPUT) character(len=*) The string which identifies the routine.

LEVEL (INPUT/OUTPUT) integer(i4b) The number of blanks to use for indentation. LEVEL is incremented by 2 on output.

TRACE (INPUT) logical(lgl) Logical indicator for printing the message.

PRT_UNIT (INPUT, OPTIONAL) integer(i4b) The printing unit.

Further Details

Trailing blanks in STRING are removed. If PRT_UNIT is absent, then all output is on the unit DEFUNIT.

6.14.4 subroutine leaving (string, level, trace, prt_unit)

Purpose

This is the 'opposite' to ENTERING. It should be called just before leaving a routine. LEVEL is reduced by 2 and if TRACE is true, an exit message is output.

Arguments

STRING (INPUT) character(len=*) The string which identifies the routine.

LEVEL (INPUT/OUTPUT) integer(i4b) The number of blanks to use for indentation. LEVEL is reduced by 2 on output.

TRACE (INPUT) logical(lgl) Logical indicator for printing the message.

PRT_UNIT (INPUT, OPTIONAL) integer(i4b) The printing unit.

Further Details

Trailing blanks in STRING are removed. If PRT_UNIT is absent, then all output is on the unit DEFUNIT.

6.14.5 subroutine indent (id, level, prt_unit)

Purpose

This is also used to indent output, albeit in a manner different from ENTERING and LEAVING. It simply writes out LEVEL blanks followed by the string ID in [], and leaves the output file marker where it is. It uses nonadvancing output. If LEVEL is not present, just the ID part is output; i.e. LEVEL is treated as zero.

Arguments

ID (INPUT) character(len=*) The string to print.

LEVEL (INPUT, OPTIONAL) integer(i4b) The number of blanks to use for indentation.

PRT_UNIT (INPUT, OPTIONAL) integer(i4b) The printing unit.

Further Details

Leading and trailing blanks in ID are removed. If PRT_UNIT is absent, then all output is on the unit DEFUNIT.

6.14.6 subroutine write_array (x, f, w, d, s, name, indent, line, prt_unit)

Purpose

Print out a real matrix with given format, as below. The matrix is printed row by row.

Print also a title for the matrix: NAME

Arguments

X (INPUT) real(stnd), dimension(:,:) The matrix to be output.

F (INPUT, OPTIONAL) character Selects the edit descriptor: FW.D . F is a character 'f', 'g', 'e' or 'd', regardless of case.

W, D (INPUT, OPTIONAL) integer(i4b) Selects the edit descriptor: fw.d . Print each entry in format FW.D . W and D are integers.

S (INPUT, OPTIONAL) integer(i4b) The number of spaces between each entry.

NAME (INPUT, OPTIONAL) character(len=*) Prints the string NAME as a title for the matrix.

IDENT (INPUT, OPTIONAL) integer(i4b) The number of blanks to use for indentation.

LINE (INPUT, OPTIONAL) integer(i4b) The number of characters per line.

PRT_UNIT (INPUT, OPTIONAL) integer(i4b) Selects the output unit.

Further Details

If the value of the argument INDENT is positive, then each output line is preceded by INDENT blank characters. If INDENT is negative, then only the matrix output will be indented (not the TITLE).

Each output line from this subroutine will have at most max(LINE,W+abs(INDENT)) characters plus one additional leading character for Fortran "carriage control". The carriage control character will always be a blank.

If LINE is absent, then line width is at most max(80,W+abs(INDENT)) characters plus one for carriage control.

All output is on the unit PRT_UNIT. If PRT_UNIT is absent, then all output is on the unit DEFUNIT.

Defaults are defined for all optional arguments. See start of module.

6.14.7 subroutine write_array (x, w, s, name, indent, line, prt_unit)

Purpose

Print out an integer matrix X with given format, as below. The matrix is printed row by row.

Print also a title for the matrix: NAME

Arguments

X (INPUT) integer(i4b), dimension(:,:) The matrix to be output.

W (INPUT, OPTIONAL) integer(i4b) Selects the width of each entry. Print each entry in format iW .

S (INPUT, OPTIONAL) integer(i4b) The number of spaces between each entry.

NAME (INPUT, OPTIONAL) character(len=*) Prints the string NAME as a title for the matrix.

IDENT (INPUT, OPTIONAL) integer(i4b) The number of blanks to use for indentation.

LINE (INPUT, OPTIONAL) integer(i4b) The number of characters per line.

PRT_UNIT (INPUT, OPTIONAL) integer(i4b) Selects the output unit.

Further Details

If W is absent, then the routine determines the best width w needed to edit the array X without excess blanks.

If the value of the argument INDENT is positive, then each output line is preceded by INDENT blank characters. If INDENT is negative, then only the matrix output will be indented (not the TITLE).

Each output line from this subroutine will have at most max(LINE,W+abs(INDENT)) characters plus one additional leading character for Fortran "carriage control". The carriage control character will always be a blank.

If LINE is absent, then line width is at most max(80,W+abs(INDENT)) characters plus one for carriage control.

All output is on the unit PRT_UNIT. If PRT_UNIT is absent, then all output is on the unit DEFUNIT.

Defaults are defined for all optional arguments. See start of module.

6.14.8 subroutine write_array (x, f, w, d, s, name, indent, line, prt_unit)

Purpose

This subroutine prints out a real vector with a given format and a title, as given in the input arguments.

Arguments

X (INPUT) real(stnd), dimension(:) The vector to be output.

F (INPUT, OPTIONAL) character Selects the edit descriptor: FW.D . F is a character 'f', 'g', 'e' or 'd', regardless of case.

W, D (INPUT, OPTIONAL) integer(i4b) Selects the edit descriptor: fw.d . Print each entry in format FW.D . W and D are integers.

S (INPUT, OPTIONAL) integer(i4b) The number of spaces between each entry.

NAME (INPUT, OPTIONAL) character(len=*) Prints the string NAME as a title for the vector.

IDENT (INPUT, OPTIONAL) integer(i4b) The number of blanks to use for indentation.

LINE (INPUT, OPTIONAL) integer(i4b) The number of characters per line.

PRT_UNIT (INPUT, OPTIONAL) integer(i4b) Selects the output unit.

Further Details

If the value of the argument INDENT is positive, then each output line is preceded by INDENT blank characters. If INDENT is negative, then only the vector output will be indented (not the TITLE).

Each output line from this subroutine will have at most max(LINE,W+abs(INDENT)) characters plus one additional leading character for Fortran "carriage control". The carriage control character will always be a blank.

If LINE is absent, then line width is at most max(80,W+abs(INDENT)) characters plus one for carriage control.

All output is on the unit PRT_UNIT. If PRT_UNIT is absent, then all output is on the unit DEFUNIT.

Defaults are defined for all optional arguments. See start of module.

6.14.9 `subroutine write_array ( x, w, s, name, indent, line, prt_unit )`

Purpose

This subroutine prints out an integer vector with a given format and a title, as given in the input arguments.

Arguments

X (INPUT) integer(i4b), dimension(:) The vector to be output.

W (INPUT, OPTIONAL) integer(i4b) Selects the width of each entry. Print each entry in format iW .

S (INPUT, OPTIONAL) integer(i4b) The number of spaces between each entry.

NAME (INPUT, OPTIONAL) character(len=*) Prints the string NAME as a title for the matrix.

IDENT (INPUT, OPTIONAL) integer(i4b) The number of blanks to use for indentation.

LINE (INPUT, OPTIONAL) integer(i4b) The number of characters per line.

PRT_UNIT (INPUT, OPTIONAL) integer(i4b) Selects the output unit.

Further Details

If W is absent, then the routine determines the best width w needed to edit the array X without excess blanks.

If the value of the argument INDENT is positive, then each output line is preceded by INDENT blank characters. If INDENT is negative, then only the vector output will be indented (not the TITLE).

Each output line from this subroutine will have at most max(line,W+abs(INDENT)) characters plus one additional leading character for Fortran "carriage control". The carriage control character will always be a blank.

If LINE is absent, then line width is at most max(80,W+abs(INDENT)) characters plus one for carriage control.

All output is on the unit PRT_UNIT. If PRT_UNIT is absent, then all output is on the unit DEFUNIT.

Defaults are defined for all optional arguments. See start of module.

6.14.10 subroutine print_array (x, f, w, d, sign_ed, s, title, namlig, namcol, indent, line, prt_unit)

Purpose

Routine for labeled real matrix output with given format, as below. The matrix is printed columns block by columns block. Print also a title for the matrix: TITLE

Arguments

X (INPUT) real(stnd), dimension(:,:) The matrix to be output.

F (INPUT, OPTIONAL) character Selects the edit descriptor: FW.D . F is a character 'f', 'g', 'e' or 'd', regardless of case.

W, D (INPUT, OPTIONAL) integer(i4b) Selects the edit descriptor: fw.d . Print each entry in format FW.D . W and D are integers.

SIGN_ED (INPUT, OPTIONAL) character(len=2) Selects the sign edit descriptor. sign_ed is 'ss' or 'sp', regardless of case.

S (INPUT, OPTIONAL) integer(i4b) the number of spaces between each entry.

TITLE (INPUT, OPTIONAL) character(len=*) Prints a title for the matrix.

NAMLIG (INPUT, OPTIONAL) character(len=*), dimension(:) Labels for the rows of the matrix. The size of NAMLIG must match the number of ligns of X.

NAMCOL (INPUT, OPTIONAL) character(len=*), dimension(:) Labels for the columns of the matrix. The size of NAMCOL must match the number of columns of X.

IDENT (INPUT, OPTIONAL) integer(i4b) The number of blanks to use for indentation.

LINE (INPUT, OPTIONAL) integer(i4b) The number of characters per line.

PRT_UNIT (INPUT, OPTIONAL) integer(i4b) Selects the output unit.

Further Details

If NAMLIG is absent, then the rows of X are labeled by row numbers.

If NAMCOL is absent, then the columns of X are labeled by column numbers. Column labels are truncated to W characters.

If the value of the argument INDENT is positive, then each output line is preceded by INDENT blank characters. If INDENT is negative, then only the matrix output will be indented (not the TITLE).

Each output line from this subroutine will have at most

max(LINE, W+abs(INDENT)+6+len_trim(NAMLIG))

characters plus one additional leading character for Fortran "carriage control". The carriage control character will always be a blank.

If LINE is absent, then line width is at most max(80, W+abs(INDENT)+6+len_trim(NAMLIG)) characters plus one for carriage control.

All output is on the unit PRT_UNIT. If PRT_UNIT is absent, then all output is on the unit DEFUNIT.

Defaults are defined for all optional arguments. See start of module.

6.14.11 subroutine print_array (x, w, sign_ed, s, title, namlig, namcol, indent, line, prt_unit)

Purpose

Routine for labeled integer matrix output with given format, as below. The matrix is printed columns block by columns block. Print also a title for the matrix: TITLE

Arguments

X (INPUT) integer(i4b), dimension(:,:) The matrix to be output.

W (INPUT, OPTIONAL) integer(i4b) Selects the width of each entry. Print each entry in format iW .

SIGN_ED (INPUT, OPTIONAL) character(len=2) Selects the sign edit descriptor. sign_ed is 'ss' or 'sp', regardless of case.

S (INPUT, OPTIONAL) integer(i4b) The number of spaces between each entry.

TITLE (INPUT, OPTIONAL) character(len=*) Prints a title for the matrix.

NAMLIG (INPUT, OPTIONAL) character(len=*), dimension(:) Labels for the rows of the matrix. The size of NAMLIG must match the number of ligns of X.

NAMCOL (INPUT, OPTIONAL) character(len=*), dimension(:) Labels for the columns of the matrix. The size of NAMCOL must match the number of columns of X.

IDENT (INPUT, OPTIONAL) integer(i4b) The number of blanks to use for indentation.

LINE (INPUT, OPTIONAL) integer(i4b) The number of characters per line.

PRT_UNIT (INPUT, OPTIONAL) integer(i4b) Selects the output unit.

Further Details

If W is absent, then the routine determines the best width w needed to edit the array X without excess blanks.

If NAMLIG is absent, then the rows of X are labeled by row numbers.

If NAMCOL is absent, then the columns of X are labeled by column numbers. Column labels are truncated to W characters.

If the value of the argument INDENT is positive, then each output line is preceded by INDENT blank characters. If INDENT is negative, then only the matrix output will be indented (not the TITLE).

Each output line from this subroutine will have at most

max(LINE, W+abs(INDENT)+6+len_trim(NAMLIG))

characters plus one additional leading character for Fortran "carriage control". The carriage control character will always be a blank.

If LINE is absent, then line width is at most max(80, W+abs(INDENT)+6+len_trim(NAMLIG)) characters plus one for carriage control.

All output is on the unit PRT_UNIT. If PRT_UNIT is absent, then all output is on the unit DEFUNIT.

Defaults are defined for all optional arguments. See start of module.

6.14.12 subroutine print_array (x, f, w, d, sign_ed, title, namlig, indent, prt_unit)

Purpose

Routine for labeled real vector output with given format, as below.

Print also a title for the vector: TITLE

Arguments

X (INPUT) real(stnd), dimension(:) The vector to be output.

F (INPUT, OPTIONAL) character Selects the edit descriptor: FW.D . F is a character 'f', 'g', 'e' or 'd', regardless of case.

W, D (INPUT, OPTIONAL) integer(i4b) Selects the edit descriptor: fw.d . Print each entry in format FW.D . W and D are integers.

SIGN_ED (INPUT, OPTIONAL) character(len=2) Selects the sign edit descriptor. sign_ed is 'ss' or 'sp'.

TITLE (INPUT, OPTIONAL) character(len=*) Prints a title for the vector.

NAMLIG (INPUT, OPTIONAL) character(len=*), dimension(:) Labels for the elements of the vector. The size of NAMLIG must match the size of X.

IDENT (INPUT, OPTIONAL) integer(i4b) The number of blanks to use for indentation.

PRT_UNIT (INPUT, OPTIONAL) integer(i4b) Selects the output unit.

Further Details

If NAMLIG is absent, then the rows of X are labeled by row numbers.

If the value of the argument INDENT is positive, then each output line is preceded by INDENT blank characters. If INDENT is negative, then only the vector output will be indented (not the TITLE).

Each output line from this subroutine will have at most

abs(INDENT) + max(len(TITLE), W+6+len_trim(NAMLIG))

characters plus one additional leading character for Fortran "carriage control". The carriage control character will always be a blank.

All output is on the unit PRT_UNIT. If PRT_UNIT is absent, then all output is on the unit DEFUNIT.

Defaults are defined for all optional arguments. See start of module.

6.14.13 subroutine print_array (x, w, sign_ed, title, namlig, indent, prt_unit)

Purpose

Routine for labeled integer vector output with given format, as below.

Print also a title for the vector: TITLE

Arguments

X (INPUT) integer(i4b), dimension(:) The vector to be output.

W (INPUT, OPTIONAL) integer(i4b) Selects the width of each entry. Print each entry in format iW .

SIGN_ED (INPUT, OPTIONAL) character(len=2) Selects the sign edit descriptor. sign_ed is 'ss' or 'sp', regardless of case.

TITLE (INPUT, OPTIONAL) character(len=*) Prints a title for the vector.

NAMLIG (INPUT, OPTIONAL) character(len=*), dimension(:) Labels for the elements of the vector. The size of NAMLIG must match the size of X.

IDENT (INPUT, OPTIONAL) integer(i4b) The number of blanks to use for indentation.

PRT_UNIT (INPUT, OPTIONAL) integer(i4b) Selects the output unit.

Further Details

If W is absent, then the routine determines the best width w needed to edit the array X without excess blanks.

If NAMLIG is absent, then the rows of X are labeled by row numbers.

If the value of the argument INDENT is positive, then each output line is preceded by INDENT blank characters. If INDENT is negative, then only the vector output will be indented (not the TITLE).

Each output line from this subroutine will have at most

abs(INDENT) + max(len(TITLE), W+6+len_trim(NAMLIG))

characters plus one additional leading character for Fortran "carriage control". The carriage control character will always be a blank.

All output is on the unit PRT_UNIT. If PRT_UNIT is absent, then all output is on the unit DEFUNIT.

Defaults are defined for all optional arguments. See start of module.

6.14.14 subroutine print_prinfac (mode, a, f, names, line, prt_unit)

Purpose

Routine for labeled matrix output after an EOF or SVD analysis.

Print an EOF model (MODE=1) or the associated principal components (MODE=2) and an SVD model (MODE=3) or the associated singular variables (MODE=4)

Arguments

MODE (INPUT) integer(i1b) integer indicator for printing.

A (INPUT) real(stnd), dimension(:,:) The matrix to be output.

F (INPUT, OPTIONAL) character Selects the edit descriptor. F is a character 'f', 'g', 'e' or 'd', regardless of case. Print each entry in format f14.6 .

NAMES (INPUT, OPTIONAL) character(len=*), dimension(:) Labels for the rows of the matrix. The size of NAMES must match the number of rows of A.

LINE (INPUT, OPTIONAL) integer(i4b) The number of characters per line.

PRT_UNIT (INPUT, OPTIONAL) integer(i4b) Selects the output unit.

Further Details

If F is absent, then the default edit descriptor is DEFF.

If NAMES is absent, then the rows of a are labeled by row numbers.

Each output line from this subroutine will have at least 20+len_trim(NAMES) characters (print at least one column of a) and at most 118+len_trim(NAMES) characters (print at most eight columns of a) plus one additional leading character for Fortran "carriage control". The carriage control character will always be a blank.

If LINE is absent, then line width is 80 characters.

If PRT_UNIT is absent, then all output is on the unit DEFUNIT.

Defaults are defined for all optional arguments. See start of module.

6.14.15 `subroutine print_prinfac ( mode, a, f, names, prt_unit )`

Purpose

Routine for labeled matrix output after an EOF or SVD analysis.

Print an EOF vector (MODE=1) or the associated principal component (MODE=2) and a singular vector (MODE=3) or the associated singular variable (MODE=4)

Arguments

MODE (INPUT) integer(i1b) integer indicator for printing.

A (INPUT) real(stnd), dimension(:) The vector to be output.

F (INPUT, OPTIONAL) character Selects the edit descriptor. F is a character 'f', 'g', 'e' or 'd', regardless of case. Print each entry in format F14.6 .

NAMES (INPUT, OPTIONAL) character(len=*), dimension(:) Labels for the elements of the vector. The size of NAMES must match the size of A.

PRT_UNIT (INPUT, OPTIONAL) integer(i4b) Selects the output unit.

Further Details

If F is absent, then the default edit descriptor is DEFF.

If NAMES is absent, then the elements of A are labeled by element numbers.

Each output line from this subroutine will have 20+len_trim(NAMES) characters plus one additional leading character for Fortran "carriage control". The carriage control character will always be a blank.

If PRT_UNIT is absent, then all output is on the unit DEFUNIT.

Defaults are defined for all optional arguments. See start of module.

6.14.16 subroutine print_stat (mode, nomiss, var, inr, qlt, names, prt_unit)

Purpose

Print statistics for an EOF "missing" analysis for

Variables (MODE=1) Observations (MODE=2)

Arguments

MODE (INPUT) integer(i1b) integer indicator for printing.

NOMISS (INPUT) integer(i4b), dimension(:) Vector giving the number of non-missing elements for each variable (MODE=1) or observation (MODE=2).

VAR, INR, QLT (INPUT) real(stnd), dimension(:) The statistics to be output for each variable (MODE=1) or observation (MODE=2). The size of these vectors must match the size of NOMISS.

NAMES (INPUT, OPTIONAL) character(len=*), dimension(:) Labels for the variables(MODE=1) or observations (MODE=2). The size of this vector must match the size of NOMISS.

PRT_UNIT (INPUT, OPTIONAL) integer(i4b) Selects the output unit.

6.14.17 subroutine print_stat (mode, weight, var, inr, qlt, names, prt_unit)

Purpose

Print statistics for an EOF "weighted" analysis for

Variables (MODE=1) Observations (MODE=2)

Arguments

MODE (INPUT) integer(i1b) integer indicator for printing.

WEIGHT (INPUT) real(stnd), dimension(:) Vector giving the weights for each variable (MODE=1) or observation (MODE=2).

VAR, INR, QLT (INPUT) real(stnd), dimension(:) The statistics to be output for each variable (MODE=1) or observation (MODE=2). The size of these vectors must match the size of WEIGHT.

NAMES (INPUT, OPTIONAL) character(len=*), dimension(:) Labels for the variables(MODE=1) or observations (MODE=2). The size of this vector must match the size of WEIGHT.

PRT_UNIT (INPUT, OPTIONAL) integer(i4b) Selects the output unit.

6.15 Module_Prob_Procedures

Copyright 2021 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING SUBROUTINES AND FUNCTIONS FOR PROBABILITY DISTRIBUTION FUNCTIONS, INVERSES, AND OTHER PARAMETERS.

LATEST REVISION : 23/08/2021

6.15.1 `function lngamma ( x )`

Purpose

Evaluates the logarithm of the gamma function: ln(gamma(X)) for a strictly positive real argument X.

Arguments

X (INPUT) real(stnd) On entry, a strictly positive real argument X.

Further Details

This function uses a Lanczos-type approximation to ln(gamma(X)) for X > 0. Its accuracy is about 14 significant digits except for small regions in the vicinity of 1 and 2.

This function is adapted from:

(1) **Lanczos, C., 1964:** A precision approximation of the gamma function. J. SIAM Numer. Anal., B, 1, 86-96.

6.15.2 `function lngamma ( x )`

Purpose

Evaluates the logarithm of the gamma function: ln(gamma(X(:))) for a strictly positive real vector argument X(:).

Arguments

X (INPUT) real(stnd), dimension(:) On entry, a strictly positive real vector argument X.

Further Details

This function uses a Lanczos-type approximation to ln(gamma(X)) for X > 0. Its accuracy is about 14 significant digits except for small regions in the vicinity of 1 and 2.

The function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Lanczos, C., 1964:** A precision approximation of the gamma function. J. SIAM Numer. Anal., B, 1, 86-96.

6.15.3 `function lngamma ( x )`

Purpose

Evaluates the logarithm of the gamma function: ln(gamma(X(:,:))) for a strictly positive real matrix argument X(:,:).

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, a strictly positive real matrix argument X.

Further Details

This function uses a Lanczos-type approximation to ln(gamma(X)) for X > 0. Its accuracy is about 14 significant digits except for small regions in the vicinity of 1 and 2.

The function is parallelized if OPENMP is used.

This function is adapted from

(1) **Lanczos, C., 1964:** A precision approximation of the gamma function. J. SIAM Numer. Anal., B, 1, 86-96.

6.15.4 `function probgamma ( x, gamp, acu, maxiter, failure )`

Purpose

Evaluates the gamma probability distribution function (e.g. the Incomplete Gamma Integral) for a positive real argument X and a strictly positive value GAMP of the parameter p of the Gamma distribution.

PROBGAMMA computes the probability that a random variable having a Gamma distribution with parameter GAMP will be less than or equal to X.

Arguments

X (INPUT) real(stnd) On entry, a positive real argument X which is the value of the upper integral limit. X must be greater or equal to zero.

GAMP (INPUT) real(stnd) On entry, a strictly positive real argument which is the value of the parameter p of the Gamma distribution. GAMP must be greater than zero.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series expansion of the incomplete Gamma integral.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

The default value is false.

Further Details

For large GAMP (e.g. GAMP>1000), this function used a normal approximation, based on the Wilson-Hilferty transformation, see reference (3), Formula 26.4.14, p.941.

Otherwise, a Pearson's series expansion is used for evaluating the integral, see reference (3), Formula 6.5.29, p.262. The "integrating" process is terminated when both the absolute and relative contributions to the integral is not greater than the value of ACU. The default value for ACU gives the maximum precision of this function.

The time taken by this function thus depends in the precision requested through ACU, and also varies slightly with the input arguments X and GAMP.

This function is more accurate than PROBGAMMA3, but it may be slower.

This function is adapted from:

(1) **Lau, C.L., 1980:** Algorithm AS 147: A simple series for the Incomplete Gamma Integral. Appl. Statist., Vol. 29, No. 1, pp. 113-114

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-squared and Incomplete Gamma Integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29 and 26.4.14). New York, Dover Publications

6.15.5 `function probgamma ( x, gamp, acu, maxiter, failure )`

Purpose

Evaluates the gamma probability distribution function (e.g. the Incomplete Gamma Integral) for a positive real vector argument X and a strictly positive value GAMP of the parameter P of the Gamma distribution.

PROBGAMMA computes the probability that a random variable having a Gamma distribution with parameter GAMP will be less than or equal to X(i) for i=1 to size(X).

Arguments

X (INPUT) real(stnd), dimension(:) On entry, a positive real vector argument X which gives the values of the upper integral limit. Elements in X(:) must be greater or equal to zero.

GAMP (INPUT) real(stnd) On entry, a strictly positive real argument which is the value of the parameter p of the Gamma distribution. GAMP must be greater than zero.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series expansion of the incomplete Gamma integral.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

The default value is false.

Further Details

For large GAMP (e.g. GAMP>1000), this function used a normal approximation, based on the Wilson-Hilferty transformation, see reference (3), Formula 26.4.14, p.941.

Otherwise, a Pearson's series expansion is used for evaluating the integral, see reference (3), Formula 6.5.29, p.262. The "integrating" process is terminated when both the absolute and relative contributions to the integral is not greater than the value of ACU. The default value for ACU gives the maximum precision of this function.

The time taken by this function thus depends in the precision requested through ACU, and also varies slightly with the input arguments X and GAMP.

This function is more accurate than PROBGAMMA3, but it may be slower.

The function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Lau, C.L., 1980:** Algorithm AS 147: A simple series for the Incomplete Gamma Integral. Appl. Statist., Vol. 29, No. 1, pp. 113-114

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-squared and Incomplete Gamma Integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29 and 26.4.14). New York, Dover Publications

6.15.6 `function probgamma ( x, gamp, acu, maxiter, failure )`

Purpose

Evaluates the gamma probability distribution function (e.g. the Incomplete Gamma Integral) for a positive real vector argument X and a strictly positive vector argument GAMP of the parameter P of the Gamma distribution.

PROBGAMMA computes the probability that a random variable having a Gamma distribution with parameter GAMP(i) will be less than or equal to X(i) for i=1 to size(X).

Arguments

X (INPUT) real(stnd), dimension(:) On entry, a positive real vector argument X which gives the values of the upper integral limit. Elements in X(:) must be greater or equal to zero.

GAMP (INPUT) real(stnd), dimension(:) On entry, a strictly positive real vector argument which are the values of the parameter p of the Gamma distribution. Elements in GAMP(:) must be greater than zero.

The size of GAMP must verify size(GAMP) = size(X) .

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series expansion of the incomplete Gamma integral.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

The default value is false.

Further Details

For large GAMP(i) (e.g. GAMP(i)>1000), this function used a normal approximation, based on the Wilson-Hilferty transformation, see reference (3), Formula 26.4.14, p.941.

Otherwise, a Pearson's series expansion is used for evaluating the integral, see reference (3), Formula 6.5.29, p.262. The "integrating" process is terminated when both the absolute and relative contributions to the integral is not greater than the value of ACU. The default value for ACU gives the maximum precision of this function.

The time taken by this function thus depends in the precision requested through ACU, and also varies slightly with the input arguments X and GAMP.

This function is more accurate than PROBGAMMA3, but it may be slower.

The function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Lau, C.L., 1980:** Algorithm AS 147: A simple series for the Incomplete Gamma Integral. Appl. Statist., Vol. 29, No. 1, pp. 113-114

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-squared and Incomplete Gamma Integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29 and 26.4.14). New York, Dover Publications

6.15.7 `function probgamma ( x, gamp, acu, maxiter, failure )`

Purpose

Evaluates the gamma probability distribution function (e.g. the Incomplete Gamma Integral) for a positive real matrix argument X and a strictly positive value GAMP of the parameter P of the Gamma distribution.

PROBGAMMA computes the probability that a random variable having a Gamma distribution with parameter GAMP will be less than or equal to X(i,j) for i=1 to size(X,1) and j=1 to size(X,2).

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, a positive real matrix argument X which gives the values of the upper integral limit. Elements in X(:,:) must be greater or equal to zero.

GAMP (INPUT) real(stnd) On entry, a strictly positive real argument which is the value of the parameter p of the Gamma distribution. GAMP must be greater than zero.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series expansion of the incomplete Gamma integral.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

The default value is false.

Further Details

For large GAMP (e.g. GAMP>1000), this function used a normal approximation, based on the Wilson-Hilferty transformation, see reference (3), Formula 26.4.14, p.941.

Otherwise, a Pearson's series expansion is used for evaluating the integral, see reference (3), Formula 6.5.29, p.262. The "integrating" process is terminated when both the absolute and relative contributions to the integral is not greater than the value of ACU. The default value for ACU gives the maximum precision of this function.

The time taken by this function thus depends in the precision requested through ACU, and also varies slightly with the input arguments X and GAMP.

This function is more accurate than PROBGAMMA3, but it may be slower.

The function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Lau, C.L., 1980:** Algorithm AS 147: A simple series for the Incomplete Gamma Integral. Appl. Statist., Vol. 29, No. 1, pp. 113-114

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-squared and Incomplete Gamma Integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29 and 26.4.14). New York, Dover Publications

6.15.8 function probgamma (x, gamp, acu, maxiter, failure)

Purpose

Evaluates the gamma probability distribution function (e.g. the Incomplete Gamma Integral) for a positive real matrix argument X and a strictly positive matrix argument GAMP of the parameter P of the Gamma distribution.

PROBGAMMA computes the probability that a random variable having a Gamma distribution with parameter GAMP(i,j) will be less than or equal to X(i,j) for i=1 to size(X,1) and j=1 to size(X,2).

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, a positive real matrix argument X which gives the values of the upper integral limit. Elements in X(:) must be greater or equal to zero.

GAMP (INPUT) real(stnd), dimension(:,:) On entry, a strictly positive real matrix argument which are the values of the parameter p of the Gamma distribution. Elements in GAMP(:,:) must be greater than zero.

The shape of GAMP must verify:

- size(GAMP,1) = size(X,1)
- size(GAMP,2) = size(X,2).

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series expansion of the incomplete Gamma integral.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

The default value is false.

Further Details

For large GAMP(i,j) (e.g. GAMP(i,j)>1000), this function used a normal approximation, based on the Wilson-Hilferty transformation, see reference (3), Formula 26.4.14, p.941.

Otherwise, a Pearson's series expansion is used for evaluating the integral, see reference (3), Formula 6.5.29, p.262. The "integrating" process is terminated when both the absolute and relative contributions to the integral is not greater than the value of ACU. The default value for ACU gives the maximum precision of this function.

The time taken by this function thus depends in the precision requested through ACU, and also varies slightly with the input arguments X and GAMP.

This function is more accurate than PROBGAMMA3, but it may be slower.

The function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Lau, C.L., 1980:** Algorithm AS 147: A simple series for the Incomplete Gamma Integral. Appl. Statist., Vol. 29, No. 1, pp. 113-114

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-squared and Incomplete Gamma Integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29 and 26.4.14). New York, Dover Publications

6.15.9 `function probgamma2 ( x, gamp, acu, maxiter, failure )`

Purpose

Evaluates the gamma probability distribution function (e.g. the Incomplete Gamma Integral) for a positive real argument X and a strictly positive value GAMP of the parameter p of the Gamma distribution.

PROBGAMMA2 computes the probability that a random variable having a Gamma distribution with parameter GAMP will be less than or equal to X.

Arguments

X (INPUT) real(stnd) On entry, a positive real argument X which is the value of the upper integral limit. X must be greater or equal to zero.

GAMP (INPUT) real(stnd) On entry, a strictly positive real argument which is the value of the parameter p of the Gamma distribution. GAMP must be greater than zero.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series or continued fraction expansion of the incomplete Gamma integral.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

The default value is false.

Further Details

For large GAMP (e.g. GAMP>1000), this function used a normal approximation, based on the Wilson-Hilferty transformation, see reference (1), Formula 26.4.14, p.941.

For X<=1 ou X<GAMP, a Pearson's series expansion is used, see reference (1), Formula 6.5.29, p.262. For other values of X, a continued fraction expansion is used since this expansion tends to converge more quickly than Pearson's series expansion, see reference (1), Formula 6.5.31, p.263. In both cases, the "integrating" process is terminated when both the absolute and relative contributions to the integral is not greater than the value of ACU. The default value for ACU gives the maximum precision of this function.

The time taken by this function thus depends in the precision requested through ACU, and also varies slightly with the input arguments X and GAMP.

This function is more accurate than PROBGAMMA3, but is slower.

This function is adapted from:

(1) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29, 6.5.31 and 26.4.14). New York, Dover Publications

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-squared and Incomplete Gamma Integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

6.15.10 `function probgamma2 ( x, gamp, acu, maxiter, failure )`

Purpose

Evaluates the gamma probability distribution function (e.g. the Incomplete Gamma Integral) for a positive real vector argument X and a strictly positive value GAMP of the parameter p of the Gamma distribution.

PROBGAMMA2 computes the probability that a random variable having a Gamma distribution with parameter GAMP will be less than or equal to X(i) for i=1 to size(X).

Arguments

X (INPUT) real(stnd), dimension(:) On entry, a positive real vector argument X which gives the values of the upper integral limit. Elements in X(:) must be greater or equal to zero.

GAMP (INPUT) real(stnd) On entry, a strictly positive real argument which is the value of the parameter p of the Gamma distribution. GAMP must be greater than zero.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series or continued fraction expansion of the incomplete Gamma integral.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

The default value is false.

Further Details

For large GAMP (e.g. GAMP>1000), this function used a normal approximation, based on the Wilson-Hilferty transformation, see reference (1), Formula 26.4.14, p.941.

For X<=1 ou X<GAMP, a Pearson's series expansion is used, see reference (1), Formula 6.5.29, p.262. For other values of X, a continued fraction expansion is used since this expansion tends to converge more quickly than Pearson's series expansion, see reference (1), Formula 6.5.31, p.263. In both cases, the

"integrating" process is terminated when both the absolute and relative contributions to the integral is not greater than the value of ACU. The default value for ACU gives the maximum precision of this function.

The time taken by this function thus depends in the precision requested through ACU, and also varies slightly with the input arguments X and GAMP.

This function is more accurate than PROBGAMMA3, but is slower.

This function is adapted from:

(1) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29, 6.5.31 and 26.4.14). New York, Dover Publications

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-squared and Incomplete Gamma Integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

6.15.11 `function probgamma2 ( x, gamp, acu, maxiter, failure )`

Purpose

Evaluates the gamma probability distribution function (e.g. the Incomplete Gamma Integral) for a positive real vector argument X and a strictly positive vector argument GAMP of the parameter P of the Gamma distribution.

PROBGAMMA2 computes the probability that a random variable having a Gamma distribution with parameter GAMP(i) will be less than or equal to X(i) for i=1 to size(X).

Arguments

X (INPUT) real(stnd), dimension(:) On entry, a positive real vector argument X which gives the values of the upper integral limit. Elements in X(:) must be greater or equal to zero.

GAMP (INPUT) real(stnd), dimension(:) On entry, a strictly positive real vector argument which are the values of the parameter p of the Gamma distribution. Elements in GAMP(:) must be greater than zero.

The size of GAMP must verify size(GAMP) = size(X) .

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series or continued fraction expansion of the incomplete Gamma integral.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

The default value is false.

Further Details

For large GAMP(i) (e.g. GAMP(i)>1000), this function used a normal approximation, based on the Wilson-Hilferty transformation, see reference (1), Formula 26.4.14, p.941.

For X(i)<=1 ou X(i)<GAMP(i), a Pearson's series expansion is used, see reference (1), Formula 6.5.29, p.262. For other values of X(i), a continued fraction expansion is used since this expansion tends to converge more quickly than Pearson's series expansion, see reference (1), Formula 6.5.31, p.263. In both cases, the "integrating" process is terminated when both the absolute and relative contributions to the integral is not greater than the value of ACU. The default value for ACU gives the maximum precision of this function.

The time taken by this function thus depends in the precision requested through ACU, and also varies slightly with the input arguments X and GAMP.

This function is more accurate than PROBGAMMA3, but is slower.

This function is adapted from:

(1) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29, 6.5.31 and 26.4.14). New York, Dover Publications

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-squared and Incomplete Gamma Integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

6.15.12 `function probgamma2 ( x, gamp, acu, maxiter, failure )`

Purpose

Evaluates the gamma probability distribution function (e.g. the Incomplete Gamma Integral) for a positive real matrix argument X and a strictly positive value GAMP of the parameter p of the Gamma distribution.

PROBGAMMA2 computes the probability that a random variable having a Gamma distribution with parameter GAMP will be less than or equal to X(i,j) for i=1 to size(X,1) and j=1 to size(X,2).

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, a positive real matrix argument X which gives the values of the upper integral limit. Elements in X(:,:) must be greater or equal to zero.

GAMP (INPUT) real(stnd) On entry, a strictly positive real argument which is the value of the parameter p of the Gamma distribution. GAMP must be greater than zero.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series or continued fraction expansion of the incomplete Gamma integral.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

The default value is false.

Further Details

For large GAMP (e.g. GAMP>1000), this function used a normal approximation, based on the Wilson-Hilferty transformation, see reference (1), Formula 26.4.14, p.941.

For X<=1 ou X<GAMP, a Pearson's series expansion is used, see reference (1), Formula 6.5.29, p.262. For other values of X, a continued fraction expansion is used since this expansion tends to converge more quickly than Pearson's series expansion, see reference (1), Formula 6.5.31, p.263. In both cases, the "integrating" process is terminated when both the absolute and relative contributions to the integral is not greater than the value of ACU. The default value for ACU gives the maximum precision of this function.

The time taken by this function thus depends in the precision requested through ACU, and also varies slightly with the input arguments X and GAMP.

This function is more accurate than PROBGAMMA3, but is slower.

The function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29, 6.5.31 and 26.4.14). New York, Dover Publications

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-squared and Incomplete Gamma Integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

6.15.13 `function probgamma2 ( x, gamp, acu, maxiter, failure )`

Purpose

Evaluates the gamma probability distribution function (e.g. the Incomplete Gamma Integral) for a positive real matrix argument X and a strictly positive matrix argument GAMP of the parameter P of the Gamma distribution.

PROBGAMMA2 computes the probability that a random variable having a Gamma distribution with parameter GAMP(i,j) will be less than or equal to X(i,j) for i=1 to size(X,1) and j=1 to size(X,2).

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, a positive real matrix argument X which gives the values of the upper integral limit. Elements in X(:,:) must be greater or equal to zero.

GAMP (INPUT) real(stnd), dimension(:,:) On entry, a strictly positive real matrix argument which are the values of the parameter p of the Gamma distribution. Elements in GAMP(:,:) must be greater than zero.

The shape of GAMP must verify:

- size(GAMP,1) = size(X,1)
- size(GAMP,2) = size(X,2).

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series or continued fraction expansion of the incomplete Gamma integral.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

The default value is false.

Further Details

For large GAMP(i,j) (e.g. GAMP(i,j)>1000), this function used a normal approximation, based on the Wilson-Hilferty transformation, see reference (1), Formula 26.4.14, p.941.

For X(i,j)<=1 ou X(i,j)<GAMP(i,j), a Pearson's series expansion is used, see reference (1), Formula 6.5.29, p.262. For other values of X(i,j), a continued fraction expansion is used since this expansion tends to converge more quickly than Pearson's series expansion, see reference (1), Formula 6.5.31, p.263. In both cases, the "integrating" process is terminated when both the absolute and relative contributions to the integral is not greater than the value of ACU. The default value for ACU gives the maximum precision of this function.

The time taken by this function thus depends in the precision requested through ACU, and also varies slightly with the input arguments X and GAMP.

This function is more accurate than PROBGAMMA3, but is slower.

The function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29, 6.5.31 and 26.4.14). New York, Dover Publications

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-squared and Incomplete Gamma Integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

6.15.14 function probgamma3 (x, gamp, acu, maxiter, failure)

Purpose

Evaluates the gamma probability distribution function (e.g. the Incomplete Gamma Integral) for a positive real argument X and a strictly positive value GAMP of the parameter p of the Gamma distribution.

PROBGAMMA3 computes the probability that a random variable having a Gamma distribution with parameter GAMP will be less than or equal to X.

Arguments

X (INPUT) real(stnd) On entry, a positive real argument X which is the value of the upper integral limit. X must be greater or equal to zero.

GAMP (INPUT) real(stnd) On entry, a strictly positive real argument which is the value of the parameter p of the Gamma distribution. GAMP must be greater than zero.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series expansion of the incomplete Gamma integral.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

The default value is false.

Further Details

For large GAMP (e.g. GAMP>1000), this function used a normal approximation, based on the Wilson-Hilferty transformation, see reference (3), Formula 26.4.14, p.941.

For X<=max(GAMP/2,13), a Pearson's series expansion is used, see reference (3), Formula 6.5.29, p.262. For larger values of X, an alternate Pearson's asymptotic series expansion is used since this expansion tends to converge more quickly, see reference (2), Formula 6.5.32, p.263.

In both cases, the "integrating" process is terminated when both the absolute and relative contributions to the integral is not greater than the value of ACU. The default value for ACU gives the maximum precision of this function.

The time taken by this function thus depends in the precision requested through ACU, and also varies slightly with the input arguments X and GAMP.

PROBGAMMA3 is faster, but less accurate than PROBGAMMA or PROBGAMMA2, since for large values of X, the alternate Pearson's series expansion is only asymptotic.

This function is adapted from:

(1) **Lau, C.L., 1980:** Algorithm AS 147: A simple series for the Incomplete Gamma Integral. Appl. Statist., Vol. 29, No. 1, pp. 113-114

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-squared and Incomplete Gamma Integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions (formulae 6.5.29, 6.5.32 and 26.4.14). New York, Dover Publications

6.15.15 function probgamma3 (x, gamp, acu, maxiter, failure)

Purpose

Evaluates the gamma probability distribution function (e.g. the Incomplete Gamma Integral) for a positive real vector argument X and a strictly positive value GAMP of the parameter P of the Gamma distribution.

PROBGAMMA3 computes the probability that a random variable having a Gamma distribution with parameter GAMP will be less than or equal to X(i) for i=1 to size(X).

Arguments

X (INPUT) real(stnd), dimension(:) On entry, a positive real vector argument X which gives the values of the upper integral limit. Elements in X(:) must be greater or equal to zero.

GAMP (INPUT) real(stnd) On entry, a strictly positive real argument which is the value of the parameter p of the Gamma distribution. GAMP must be greater than zero.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series expansion of the incomplete Gamma integral.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

The default value is false.

Further Details

For large GAMP (e.g. GAMP>1000), this function used a normal approximation, based on the Wilson-Hilferty transformation, see reference (3), Formula 26.4.14, p.941.

For X<=max(GAMP/2,13), a Pearson's series expansion is used, see reference (3), Formula 6.5.29, p.262. For larger values of X, an alternate Pearson's asymptotic series expansion is used since this expansion tends to converge more quickly, see reference (2), Formula 6.5.32, p.263.

In both cases, the "integrating" process is terminated when both the absolute and relative contributions to the integral is not greater than the value of ACU. The default value for ACU gives the maximum precision of this function.

The time taken by this function thus depends in the precision requested through ACU, and also varies slightly with the input arguments X and GAMP.

PROBGAMMA3 is faster, but less accurate than PROBGAMMA or PROBGAMMA2, since for large values of X, the alternate Pearson's series expansion is only asymptotic.

This function is adapted from:

(1) **Lau, C.L., 1980:** Algorithm AS 147: A simple series for the Incomplete Gamma Integral. Appl. Statist., Vol. 29, No. 1, pp. 113-114

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-squared and Incomplete Gamma Integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions (formulae 6.5.29, 6.5.32 and 26.4.14). New York, Dover Publications

6.15.16 `function probgamma3 ( x, gamp, acu, maxiter, failure )`

Purpose

Evaluates the gamma probability distribution function (e.g. the Incomplete Gamma Integral) for a positive real vector argument X and a strictly positive vector argument GAMP of the parameter P of the Gamma distribution.

PROBGAMMA3 computes the probability that a random variable having a Gamma distribution with parameter GAMP(i) will be less than or equal to X(i) for i=1 to size(X).

Arguments

X (INPUT) real(stnd), dimension(:) On entry, a positive real vector argument X which gives the values of the upper integral limit. Elements in X(:) must be greater or equal to zero.

GAMP (INPUT) real(stnd), dimension(:) On entry, a strictly positive real vector argument which are the values of the parameter p of the Gamma distribution. Elements in GAMP(:) must be greater than zero.

The size of GAMP must verify size(GAMP) = size(X) .

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series expansion of the incomplete Gamma integral.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

The default value is false.

Further Details

For large GAMP(i) (e.g. GAMP(i)>1000), this function used a normal approximation, based on the Wilson-Hilferty transformation, see reference (3), Formula 26.4.14, p.941.

For X(i)<=max(GAMP(i)/2,13), a Pearson's series expansion is used, see reference (3), Formula 6.5.29, p.262. For larger values of X(i), an alternate Pearson's asymptotic series expansion is used since this expansion tends to converge more quickly, see reference (2), Formula 6.5.32, p.263.

In both cases, the "integrating" process is terminated when both the absolute and relative contributions to the integral is not greater than the value of ACU. The default value for ACU gives the maximum precision of this function.

The time taken by this function thus depends in the precision requested through ACU, and also varies slightly with the input arguments X and GAMP.

PROBGAMMA3 is faster, but less accurate than PROBGAMMA or PROBGAMMA2, since for large values of X, the alternate Pearson's series expansion is only asymptotic.

This function is adapted from:

(1) **Lau, C.L., 1980:** Algorithm AS 147: A simple series for the Incomplete Gamma Integral. Appl. Statist., Vol. 29, No. 1, pp. 113-114

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-squared and Incomplete Gamma Integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions (formulae 6.5.29, 6.5.32 and 26.4.14). New York, Dover Publications

6.15.17 `function probgamma3 ( x, gamp, acu, maxiter, failure )`

Purpose

Evaluates the gamma probability distribution function (e.g. the Incomplete Gamma Integral) for a positive real matrix argument X and a strictly positive value GAMP of the parameter P of the Gamma distribution.

PROBGAMMA3 computes the probability that a random variable having a Gamma distribution with parameter GAMP will be less than or equal to X(i,j) for i=1 to size(X,1) and j=1 to size(X,2).

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, a positive real matrix argument X which gives the values of the upper integral limit. Elements in X(:,:) must be greater or equal to zero.

GAMP (INPUT) real(stnd) On entry, a strictly positive real argument which is the value of the parameter p of the Gamma distribution. GAMP must be greater than zero.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series expansion of the incomplete Gamma integral.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

The default value is false.

Further Details

For large GAMP (e.g. GAMP>1000), this function used a normal approximation, based on the Wilson-Hilferty transformation, see reference (3), Formula 26.4.14, p.941.

For X<=max(GAMP/2,13), a Pearson's series expansion is used, see reference (3), Formula 6.5.29, p.262. For larger values of X, an alternate Pearson's asymptotic series expansion is used since this expansion tends to converge more quickly, see reference (2), Formula 6.5.32, p.263.

In both cases, the "integrating" process is terminated when both the absolute and relative contributions to the integral is not greater than the value of ACU. The default value for ACU gives the maximum precision of this function.

The time taken by this function thus depends in the precision requested through ACU, and also varies slightly with the input arguments X and GAMP.

PROBGAMMA3 is faster, but less accurate than PROBGAMMA or PROBGAMMA2, since for large values of X, the alternate Pearson's series expansion is only asymptotic.

The function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Lau, C.L., 1980:** Algorithm AS 147: A simple series for the Incomplete Gamma Integral. Appl. Statist., Vol. 29, No. 1, pp. 113-114

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-squared and Incomplete Gamma Integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions (formulae 6.5.29, 6.5.32 and 26.4.14). New York, Dover Publications

6.15.18 `function probgamma3 ( x, gamp, acu, maxiter, failure )`

Purpose

Evaluates the gamma probability distribution function (e.g. the Incomplete Gamma Integral) for a positive real matrix argument X and a strictly positive matrix argument GAMP of the parameter P of the Gamma distribution.

PROBGAMMA3 computes the probability that a random variable having a Gamma distribution with parameter GAMP(i,j) will be less than or equal to X(i,j) for i=1 to size(X,1) and j=1 to size(X,2).

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, a positive real matrix argument X which gives the values of the upper integral limit. Elements in X(:,:) must be greater or equal to zero.

GAMP (INPUT) real(stnd), dimension(:,:) On entry, a strictly positive real matrix argument which are the values of the parameter p of the Gamma distribution. Elements in GAMP(:,:) must be greater than zero.

The shape of GAMP must verify:

- size(GAMP,1) = size(X,1)
- size(GAMP,2) = size(X,2).

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series expansion of the incomplete Gamma integral.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

The default value is false.

Further Details

For large GAMP(i,j) (e.g. GAMP(i,j)>1000), this function used a normal approximation, based on the Wilson-Hilferty transformation, see reference (3), Formula 26.4.14, p.941.

For X(i,j)<=max(GAMP(i,j)/2,13), a Pearson's series expansion is used, see reference (3), Formula 6.5.29, p.262. For larger values of X(i,j), an alternate Pearson's asymptotic series expansion is used since this expansion tends to converge more quickly, see reference (2), Formula 6.5.32, p.263.

In both cases, the "integrating" process is terminated when both the absolute and relative contributions to the integral is not greater than the value of ACU. The default value for ACU gives the maximum precision of this function.

The time taken by this function thus depends in the precision requested through ACU, and also varies slightly with the input arguments X and GAMP.

PROBGAMMA3 is faster, but less accurate than PROBGAMMA or PROBGAMMA2, since for large values of X, the alternate Pearson's series expansion is only asymptotic.

The function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Lau, C.L., 1980:** Algorithm AS 147: A simple series for the Incomplete Gamma Integral. Appl. Statist., Vol. 29, No. 1, pp. 113-114

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-squared and Incomplete Gamma Integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions (formulae 6.5.29, 6.5.32 and 26.4.14). New York, Dover Publications

6.15.19 `function pinvgamma ( p, gamp, acu, maxiter )`

Purpose

Evaluates the inverse gamma probability distribution function.

For given arguments P (0<=P<=1) and GAMP (GAMP>0), PINVGAMMA returns the value X such that P is the probability that a random variable distributed as a gamma distribution with parameter GAMP is less than or equal to X.

Arguments

P (INPUT) real(stnd) On entry, input probability. P must be in the range (0,1) inclusive.

GAMP (INPUT) real(stnd) on entry, the parameter p of the gamma distribution. GAMP must be greater or equal to 0.25 .

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result when computing the incomplete Gamma integral in the evaluation of the seven term Taylor series in function PINVQ2 .

If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series or continued fraction expansion of the incomplete Gamma integral.

The default value is 1000.

See the description of the PROBGAMMA2 function for more details on this argument.

Further Details

This function actually uses PINVQ2 function and is adapted from:

(1) **Best, D.J., and Roberts, D.E., 1975:** Algorithm AS 91: The Percentage Points of the chi2 Distribution. Appl. Statist., Vol.24, No. 3, pp.385-388

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-squared and Incomplete Gamma Integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Shea, B.L., 1991:** Algorithm AS R85: A remark on AS 91: The Percentage Points of the chi2 Distribution. Appl. Statist. Vol.40, No. 1, pp.233-235.

6.15.20 `function probbeta ( x, a, b, beta, acu, maxiter, failure )`

Purpose

Evaluates the beta probability distribution function (e.g the incomplete beta function).

For given arguments X (0<=X<=1), A (A>0), B (B>0), PROBBETA returns the probability that a random variable from a beta distribution having parameters A and B will be less than or equal to X.

Arguments

X (INPUT) real(stnd) On entry, the value to which function is to be integrated. X must be in range (0,1) inclusive.

A (INPUT) real(stnd) on entry, the (1st) parameter of the beta distribution. A must be greater than 0.

B (INPUT) real(stnd) On entry, the (2nd) parameter of the beta distribution. B must be greater than 0.

BETA (INPUT, OPTIONAL) real(stnd) On entry, the logarithm of the complete beta function BETA(A,B).

If BETA is not given, the logarithm of the beta function is computed with the help of function LNGAMMA.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result. The "integrating" process is terminated when both the absolute and relative contributions to the integral is not greater than the value of ACU.

ACU is a small strictly positive integer. If the number of decimal digits' accuracy required is r, ACU should be set to 10**(-(r+1)).

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the power series representation of the incomplete Beta integral.

The default value is 2000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

The default value is false.

Further Details

This function is adapted from:

(1) **Majumder, K.L., and Bhattacharjee, G.P., 1973:** Algorithm AS 63: the Incomplete Beta Integral. Appl. Statist., Vol.22, No.3, pp 409-411.

(2) **Cran, G.W., Martin, K.J., and Thomas, G.E., 1977:** Remark AS R19 and Algorithm AS 109: A remark on Algorithms: AS 63 the Incomplete Beta integral, AS 64 Inverse of the Incomplete Beta Function Ratio. Appl. Statist., Vol.26, No.1, pp 111-114.

6.15.21 `function probbeta ( x, a, b, beta, acu, maxiter, failure )`

Purpose

Evaluates the beta probability distribution function (e.g the incomplete beta function).

For given arguments X(:) (0<=X(:)<=1), A (A>0), B (B>0), PROBBETA returns the probabilities that a random variable from a beta distribution having parameters A and B will be less than or equal to X(:).

Arguments

X (INPUT) real(stnd), dimension(:) On entry, the values to which function is to be integrated. Elements in X(:) must be in the range (0,1) inclusive.

A (INPUT) real(stnd) on entry, the (1st) parameter of the beta distribution. A must be greater than 0.

B (INPUT) real(stnd) On entry, the (2nd) parameter of the beta distribution. B must be greater than 0.

BETA (INPUT, OPTIONAL) real(stnd) On entry, the logarithm of the complete beta function BETA(A,B).

If BETA is not given, the logarithm of the beta function is computed with the help of function LNGAMMA.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result. The "integrating" process is terminated when both the absolute and relative contributions to the integral is not greater than the value of ACU.

ACU is a small strictly positive integer. If the number of decimal digits' accuracy required is r, ACU should be set to 10**(-(r+1)).

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the power series representation of the incomplete Beta integral.

The default value is 2000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

The default value is false.

Further Details

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Majumder, K.L., and Bhattacharjee, G.P., 1973:** Algorithm AS 63: the Incomplete Beta Integral. Appl. Statist., Vol.22, No.3, pp 409-411.

(2) **Cran, G.W., Martin, K.J., and Thomas, G.E., 1977:** Remark AS R19 and Algorithm AS 109: A remark on Algorithms: AS 63 the Incomplete Beta integral, AS 64 Inverse of the Incomplete Beta Function Ratio. Appl. Statist., Vol.26, No.1, pp 111-114.

6.15.22 `function probbeta ( x, a, b, beta, acu, maxiter, failure )`

Purpose

Evaluates the beta probability distribution function (e.g the incomplete beta function).

For given arguments X(:,:) (0<=X(:,:)<=1), A (A>0), B (B>0), PROBBETA returns the probabilities that a random variable from a beta distribution having parameters A and B will be less than or equal to X(:,:).

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, the values to which function is to be integrated. Elements in X(:,:) must be in the range (0,1) inclusive.

A (INPUT) real(stnd) on entry, the (1st) parameter of the beta distribution. A must be greater than 0.

B (INPUT) real(stnd) On entry, the (2nd) parameter of the beta distribution. B must be greater than 0.

BETA (INPUT, OPTIONAL) real(stnd) On entry, the logarithm of the complete beta function BETA(A,B).

If BETA is not given, the logarithm of the beta function is computed with the help of function LNGAMMA.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result. The "integrating" process is terminated when both the absolute and relative contributions to the integral is not greater than the value of ACU.

ACU is a small strictly positive integer. If the number of decimal digits' accuracy required is r, ACU should be set to 10**(-(r+1)).

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the power series representation of the incomplete Beta integral.

The default value is 2000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

The default value is false.

Further Details

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Majumder, K.L., and Bhattacharjee, G.P., 1973:** Algorithm AS 63: the Incomplete Beta Integral. Appl. Statist., Vol.22, No.3, pp 409-411.

(2) **Cran, G.W., Martin, K.J., and Thomas, G.E., 1977:** Remark AS R19 and Algorithm AS 109: A remark on Algorithms: AS 63 the Incomplete Beta integral, AS 64 Inverse of the Incomplete Beta Function Ratio. Appl. Statist., Vol.26, No.1, pp 111-114.

6.15.23 `function pinvbeta ( p, a, b, beta, acu, maxiter )`

Purpose

Evaluates the inverse beta probability distribution function (e.g. the incomplete beta function).

For given arguments P (0<=P<=1), A (A>0.1), B (B>0.1), PINVBETA returns the value X such that P is the probability that a random variable distributed as beta(A,B) is less than or equal to X.

Arguments

P (INPUT) real(stnd) On entry, input probability. P must be in the range (0,1) inclusive.

A (INPUT) real(stnd) on entry, the (1st) parameter of the beta distribution. A must be greater than 0.1 .

B (INPUT) real(stnd) On entry, the (2nd) parameter of the beta distribution. B must be greater than 0.1 .

BETA (INPUT, OPTIONAL) real(stnd) On entry, the logarithm of the complete beta function beta(A,B).

If BETA is not given, the logarithm of the beta function is computed with the help of function LNGAMMA.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result, when computing the incomplete beta function. The "integrating" process for evaluating the incomplete beta function is terminated when the relative contribution to the integral is not greater than the value of ACU. ACU is a small strictly positive integer.

See the description of the PROBBETA function for more details on this argument.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the power series representation of the incomplete Beta integral.

See the description of the PROBBETA function for more details on this argument.

The default value is 2000.

Further Details

This function is not very accurate for small values of A and/or B (e.g. less than 0.5).

This function is adapted from:

(1) **Majumder, K.L., and Bhattacharjee, G.P., 1973:** Algorithm AS 64: Inverse of the Incomplete Beta Function Ratio. Appl. Statist., Vol.22, No.3, pp 411-414.

(2) **Cran, G.W., Martin, K.J., and Thomas, G.E., 1977:** Remark AS R19 and Algorithm AS 109: A remark on Algorithms: AS 63 the Incomplete Beta integral, AS 64 Inverse of the Incomplete Beta Function Ratio. Appl. Statist., Vol.26, No.1, pp 111-114.

(3) **Berry, K.J., Mielke, P.W., and Cran, G.W., 1990:** Algorithm AS R83: A remark on Algorithm AS 109: Inverse of the Incomplete Beta Function Ratio. Appl. Statist., Vol.39, No 2, pp. 309-310.

(4) **Berry, K.J., Mielke, P.W., and Cran, G.W., 1991:** Correction to Algorithm AS R83: A remark on Algorithm AS 109: Inverse of the Incomplete Beta Function Ratio. Appl. Statist. Vol. 40, No. 1, p.236

6.15.24 `function probn ( x, upper )`

Purpose

Evaluates the standard normal (Gaussian) distribution function from X to infinity if UPPER is true or from minus infinity to X if UPPER is false. In other words, if:

- UPPER = true : PROBN = prob(U > X) ,
- UPPER = false : PROBN = prob(U < X) ,

, for U = Laplace_Gauss(0;1).

Arguments

X (INPUT) real(stnd) On entry, upper or lower limit of integration.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X is calculated.
- UPPER = false : probability to the left of X is calculated.

Further Details

This function is adapted from:

(1) **Hill, I.D., 1973:** Algorithm AS66: The Normal Integral. Applied Statistics, vol.22, no.3, pp.424-427

6.15.25 `function probn ( x, upper )`

Purpose

Evaluates the standard normal (Gaussian) distribution function from X(i) to infinity if UPPER is true or from minus infinity to X(i) if UPPER is false, for i=1 to size(X). In other words, if:

- UPPER = true : PROBN(i) = prob(U > X(i)) ,
- UPPER = false : PROBN(i) = prob(U < X(i)) ,

, for U = Laplace_Gauss(0;1) and i=1 to size(X).

Arguments

X (INPUT) real(stnd), dimension(:) On entry, upper or lower limits of integration.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X is calculated.
- UPPER = false : probability to the left of X is calculated.

Further Details

The subroutine is parallelized if OPENMP is used.

This function is adapted from:

(1) **Hill, I.D., 1973:** Algorithm AS66: The Normal Integral. Applied Statistics, vol.22, no.3, pp.424-427

6.15.26 `function probn ( x, upper )`

Purpose

Evaluates the standard normal (Gaussian) distribution function from X(i,j) to infinity if UPPER is true or from minus infinity to X(i,j) if UPPER is false, for i=1 to size(X,1) and j=1 to size(X,2). In other words, if:

- UPPER = true : PROBN(i, j) = prob(U > X(i,j)) ,
- UPPER = false : PROBN(i, j) = prob(U < X(i,j)) ,

, for U = Laplace_Gauss(0;1) and i=1 to size(X,1) and j=1 to size(X,2).

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, upper or lower limits of integration.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X is calculated.
- UPPER = false : probability to the left of X is calculated.

Further Details

The subroutine is parallelized if OPENMP is used.

This function is adapted from:

(1) **Hill, I.D., 1973:** Algorithm AS66: The Normal Integral. Applied Statistics, vol.22, no.3, pp.424-427

6.15.27 function probn2 (x, upper)

Purpose

Evaluates the standard normal (Gaussian) distribution function from X to infinity if UPPER is true or from minus infinity to X if UPPER is false. In other words, if:

- UPPER = true : PROBN2 = prob(U > X) ,
- UPPER = false : PROBN2 = prob(U < X) ,

, for U = Laplace_Gauss(0;1).

Arguments

X (INPUT) real(extd) On entry, upper or lower limit of integration.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X is calculated.
- UPPER = false : probability to the left of X is calculated.

Further Details

This function gives higher accuracy than PROBN.

This function is based upon algorithm 5666 for the error function, from:

(1) **Hart, J.F. et al, 1968:** Computer Approximations. 354 pp, New York, Wiley.

6.15.28 function probn2 (x, upper)

Purpose

Evaluates the standard normal (Gaussian) distribution function from X(i) to infinity if UPPER is true or from minus infinity to X(i) if UPPER is false, for i=1 to size(X). In other words, if:

- UPPER = true : PROBN2(i) = prob(U > X(i)) ,
- UPPER = false : PROBN2(i) = prob(U < X(i)) ,

, for U = Laplace_Gauss(0;1) and i=1 to size(X).

Arguments

X (INPUT) real(extd), dimension(:) On entry, upper or lower limits of integration.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X is calculated.
- UPPER = false : probability to the left of X is calculated.

Further Details

The subroutine is parallelized if OPENMP is used.

This function gives higher accuracy than PROBN.

This function is based upon algorithm 5666 for the error function, from:

(1) **Hart, J.F. et al, 1968:** Computer Approximations. 354 pp, New York, Wiley.

6.15.29 function probn2 (x, upper)

Purpose

Evaluates the standard normal (Gaussian) distribution function from X(i,j) to infinity if UPPER is true or from minus infinity to X(i,j) if UPPER is false, for i=1 to size(X,1) and j=1 to size(X,2). In other words, if:

- UPPER = true : PROBN2(i, j) = prob(U > X(i,j)) ,
- UPPER = false : PROBN2(i, j) = prob(U < X(i,j)) ,

, for U = Laplace_Gauss(0;1) and i=1 to size(X,1) and j=1 to size(X,2).

Arguments

X (INPUT) real(extd), dimension(:,:) On entry, upper or lower limits of integration.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X is calculated.
- UPPER = false : probability to the left of X is calculated.

Further Details

The subroutine is parallelized if OPENMP is used.

This function gives higher accuracy than PROBN.

This function is based upon algorithm 5666 for the error function, from:

(1) **Hart, J.F. et al, 1968:** Computer Approximations. 354 pp, New York, Wiley.

6.15.30 function pinvn (p)

Purpose

Evaluates the inverse of the standard normal (Gaussian) distribution function:

X0 = PINVN(P)

, if P = probability(U < X0) for U = Laplace_Gauss(0;1).

PINVN returns the normal deviate X0 corresponding to a given lower tail area of P.

Arguments

P (INPUT) real(stnd) On entry, the probability. P must verify 0. < P < 1.

Further Details

The inverse Gaussian CDF is approximated to high precision using rational approximations (polynomials with degree 2 and 3) by the subroutine PPND7 described in the reference (1).

This function is accurate to about seven decimal figures for min(P,1-P) > 10**(-316).

If P is very close to unity, a serious loss of significance may be incurred in forming 1 - P = c in the code of the function. In this circumstance the user should, if possible, evaluate c directly or in extended precision and evaluate X0(P) = PINVN(P) as -X0(c) = PINVN(c).

The hash sums below are the sums of the mantissas of the coefficients. They are included for use in checking transcription.

This function is adapted from the routine PPND7 described in:

(1) **Wichura, M.J., 1988:** Algorithm AS 241: The percentage points of the normal distribution. Appl. Statist., Vol. 37, No. 3, pp. 477-484.

6.15.31 `function pinvn ( p )`

Purpose

Evaluates the inverse of the standard normal (Gaussian) distribution function:

X0(i) = PINVN(P(i))

, if P(i) = probability(U < X0(i)) for U = Laplace_Gauss(0;1) and i=1 to size(P).

PINVN returns the normal deviate X0(i) corresponding to a given lower tail area of P(i), for i=1 to size(P).

Arguments

P (INPUT) real(stnd), dimension(:) On entry, the probabilities. P(i) must verify 0. < P(i) < 1, for i=1 to size(P) .

Further Details

The inverse Gaussian CDF is approximated to high precision using rational approximations (polynomials with degree 2 and 3) by the subroutine PPND7 described in the reference (1).

This function is accurate to about seven decimal figures for min(P,1-P) > 10**(-316).

If P is very close to unity, a serious loss of significance may be incurred in forming 1 - P = c in the code of the function. In this circumstance the user should, if possible, evaluate c directly or in extended precision and evaluate X0(P) = PINVN(P) as -X0(c) = PINVN(c).

The subroutine is parallelized if OPENMP is used.

The hash sums below are the sums of the mantissas of the coefficients. They are included for use in checking transcription.

This function is adapted from the routine PPND7 described in:

(1) **Wichura, M.J., 1988:** Algorithm AS 241: The percentage points of the normal distribution. Appl. Statist., Vol. 37, No. 3, pp. 477-484.

6.15.32 function pinvn (p)

Purpose

Evaluates the inverse of the standard normal (Gaussian) distribution function:

X0(i,j) = PINVN(P(i,j))

, if P(i,j) = probability(U < X0(i,j)) for U = Laplace_Gauss(0;1), i=1 to size(P,1) and j=1 to size(P,2).

PINVN returns the normal deviate X0(i,j) corresponding to a given lower tail area of P(i,j) for i=1 to size(P,1) and j=1 to size(P,2).

Arguments

P (INPUT) real(stnd), dimension(:,:) On entry, the probabilities. P(i,j) must verify 0. < P(i,j) < 1, for i=1 to size(P,1) and j=1 to size(P,2) .

Further Details

The inverse Gaussian CDF is approximated to high precision using rational approximations (polynomials with degree 2 and 3) by the subroutine PPND7 described in the reference (1).

This function is accurate to about seven decimal figures for min(P,1-P) > 10**(-316).

If P is very close to unity, a serious loss of significance may be incurred in forming 1 - P = c in the code of the function. In this circumstance the user should, if possible, evaluate c directly or in extended precision and evaluate X0(P) = PINVN(P) as -X0(c) = PINVN(c).

The subroutine is parallelized if OPENMP is used.

The hash sums below are the sums of the mantissas of the coefficients. They are included for use in checking transcription.

This function is adapted from the routine PPND7 described in:

(1) **Wichura, M.J., 1988:** Algorithm AS 241: The percentage points of the normal distribution. Appl. Statist., Vol. 37, No. 3, pp. 477-484.

6.15.33 function pinvn2 (p)

Purpose

Evaluates the inverse of the standard normal (Gaussian) distribution function:

X0 = PINVN2(P)

, if P = probability(U < X0) for U = Laplace_Gauss(0;1).

PINVN2 returns the normal deviate X0 corresponding to a given lower tail area of P.

Arguments

P (INPUT) real(extd) On entry, the probability. P must verify 0. < P < 1.

Further Details

The inverse Gaussian CDF is approximated to high precision using rational approximations (polynomials with degree 7) by the subroutine PPND16 described in the reference (1).

This function is accurate to about 16 decimal figures for min(P,1-P) > 10**(-316) and gives higher accuracy than PINVN function, but it is slower.

On a machine, that uses only 32 bits to represent real variables, PINVN2 should be implemented in double precision (e.g. with a correct choice of the kind EXTD in the module Select_Parameters).

If P is very close to unity, a serious loss of significance may be incurred in forming 1 - P = c in the code of the function. In this circumstance the user should, if possible, evaluate c directly or in extended precision and evaluate X0(P) = PINVN2(P) as -X0(c) = PINVN2(c) .

The hash sums below are the sums of the mantissas of the coefficients. They are included for use in checking transcription.

This function is adapted from the routine PPND16 described in:

(1) **Wichura, M.J., 1988:** Algorithm AS 241: The percentage points of the normal distribution. Appl. Statist., Vol. 37, No. 3, pp. 477-484.

6.15.34 `function pinvn2 ( p )`

Purpose

Evaluates the inverse of the standard normal (Gaussian) distribution function:

X0(i) = PINVN2(P(i))

, if P(i) = probability(U < X0(i)) for U = Laplace_Gauss(0;1) and i=1 to size(P).

PINVN returns the normal deviate X0(i) corresponding to a given lower tail area of P(i), for i=1 to size(P).

Arguments

P (INPUT) real(extd), dimension(:) On entry, the probabilities. P(i) must verify 0. < P(i) < 1, for i=1 to size(P) .

Further Details

The inverse Gaussian CDF is approximated to high precision using rational approximations (polynomials with degree 7) by the subroutine PPND16 described in the reference (1).

This function is accurate to about 16 decimal figures for min(P,1-P) > 10**(-316) and gives higher accuracy than PINVN function, but it is slower.

On a machine, that uses only 32 bits to represent real variables, PINVN2 should be implemented in double precision (e.g. with a correct choice of the kind EXTD in the module Select_Parameters).

If P is very close to unity, a serious loss of significance may be incurred in forming 1 - P = c in the code of the function. In this circumstance the user should, if possible, evaluate c directly or in extended precision and evaluate X0(P) = PINVN2(P) as -X0(c) = PINVN2(c).

The subroutine is parallelized if OPENMP is used.

The hash sums below are the sums of the mantissas of the coefficients. They are included for use in checking transcription.

This function is adapted from the routine PPND16 described in:

(1) **Wichura, M.J., 1988:** Algorithm AS 241: The percentage points of the normal distribution. Appl. Statist., Vol. 37, No. 3, pp. 477-484.

6.15.35 `function pinvn2 ( p )`

Purpose

Evaluates the inverse of the standard normal (Gaussian) distribution function:

X0(i,j) = PINVN2(P(i,j))

, if P(i,j) = probability(U < X0(i,j)) for U = Laplace_Gauss(0;1), i=1 to size(P,1) and j=1 to size(P,2).

PINVN2 returns the normal deviate X0(i,j) corresponding to a given lower tail area of P(i,j) for i=1 to size(P,1) and j=1 to size(P,2).

Arguments

P (INPUT) real(extd), dimension(:,:) On entry, the probabilities. P(i,j) must verify 0. < P(i,j) < 1, for i=1 to size(P,1) and j=1 to size(P,2) .

Further Details

The inverse Gaussian CDF is approximated to high precision using rational approximations (polynomials with degree 7) by the subroutine PPND16 described in the reference (1).

This function is accurate to about 16 decimal figures for min(P,1-P) > 10**(-316) and gives higher accuracy than PINVN function, but it is slower.

On a machine, that uses only 32 bits to represent real variables, PINVN2 should be implemented in double precision (e.g. with a correct choice of the kind EXTD in the module Select_Parameters).

If P is very close to unity, a serious loss of significance may be incurred in forming 1 - P = c in the code of the function. In this circumstance the user should, if possible, evaluate c directly or in extended precision and evaluate X0(P) = PINVN2(P) as -X0(c) = PINVN2(c).

The subroutine is parallelized if OPENMP is used.

The hash sums below are the sums of the mantissas of the coefficients. They are included for use in checking transcription.

This function is adapted from the routine PPND16 described in:

(1) **Wichura, M.J., 1988:** Algorithm AS 241: The percentage points of the normal distribution. Appl. Statist., Vol. 37, No. 3, pp. 477-484.

6.15.36 function probt (t, ndf, upper, ndf_max)

Purpose

Evaluates the Student's t-distribution function from T to infinity if UPPER is true or from minus infinity to T if UPPER is false. In otherwords, if:

- UPPER = true : PROBT = prob(U > T) ,
- UPPER = false : PROBT = prob(U < T) ,

, for U = Student(NDF).

Arguments

T (INPUT) real(stnd) On entry, upper or lower limit of integration of the t-density.

NDF (INPUT) integer(i4b) On entry, degrees of freedom of the t-distribution. NDF must be greater or equal to 1.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of T is calculated.
- UPPER = false : probability to the left of T is calculated.

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, if:

- NDF is lower or equal to NDF_MAX, the t_density is integrated.
- NDF is greater than NDF_MAX, an asymptotic series is used.

The default is 20.

Further Details

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-distribution (Algorithm 395). Comm. A.C.M., vol.13, 617-619

(2) **Cooper, B.E., 1968:** The integral of student's t distribution, (Algorithm AS3). Applied Statistics, vol.17, no.2, 189

6.15.37 function probt (t, ndf, upper, ndf_max)

Purpose

Evaluates the Student's t-distribution function from T(i) to infinity if UPPER is true or from minus infinity to T(i) if UPPER is false, for i=1 to size(T). In other words, if:

- UPPER = true : PROBT(i) = prob(U > T(i)) ,
- UPPER = false : PROBT(i) = prob(U < T(i)) ,

, for U = STUDENT(NDF) and i=1 to size(T).

Arguments

T (INPUT) real(stnd), dimension(:) On entry, upper or lower limits of integration of the t-density.

NDF (INPUT) integer(i4b) On entry, degrees of freedom of the t-distribution. NDF must be greater or equal to 1.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probabilities to the right of T is calculated.
- UPPER = false : probabilities to the left of T is calculated.

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, if:

- NDF is lower or equal to NDF_MAX, the t_density is integrated.
- NDF is greater than NDF_MAX, an asymptotic series is used.

The default is 20.

Further Details

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-distribution (Algorithm 395). Comm. A.C.M., vol.13, 617-619

(2) **Cooper, B.E., 1968:** The integral of student's t distribution, (Algorithm AS3). Applied Statistics, vol.17, no.2, 189

6.15.38 `function probt ( t, ndf, upper, ndf_max )`

Purpose

Evaluates the Student's t-distribution function from T(i) to infinity if UPPER is true or from minus infinity to T(i) if UPPER is false, for i=1 to size(T). In other words, if:

- UPPER = true : PROBT(i) = prob(U > T(i)) ,
- UPPER = false : PROBT(i) = prob(U < T(i)) ,

, for U = STUDENT(NDF(i)) and i=1 to size(T).

Arguments

T (INPUT) real(stnd), dimension(:) On entry, upper or lower limits of integration of the t-density.

NDF (INPUT) integer(i4b), dimension(:) On entry, degrees of freedom of the t-distribution. Any value in the array NDF must be greater or equal to 1.

The size of NDF must be size(NDF) = size(T) .

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probabilities to the right of T is calculated.
- UPPER = false : probabilities to the left of T is calculated.

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, if:

- NDF is lower or equal to NDF_MAX, the t_density is integrated.
- NDF is greater than NDF_MAX, an asymptotic series is used.

The default is 20.

Further Details

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-distribution (Algorithm 395). Comm. A.C.M., vol.13, 617-619

(2) **Cooper, B.E., 1968:** The integral of student's t distribution, (Algorithm AS3). Applied Statistics, vol.17, no.2, 189

6.15.39 `function probt ( t, ndf, upper, ndf_max )`

Purpose

Evaluates the Student's t-distribution function from T(i,j) to infinity if UPPER is true or from minus infinity to T(i,j) if UPPER is false, for i=1 to size(T,1) and j=1 to size(T,2). In other words, if:

- UPPER = true : PROBT(i, j) = prob(U > T(i,j)) ,
- UPPER = false : PROBT(i, j) = prob(U < T(i,j)) ,

, for U = Student(NDF), i=1 to size(T,1) and j=1 to size(T,2).

Arguments

T (INPUT) real(stnd), dimension(:,:) On entry, upper or lower limits of integration of the t-density.

NDF (INPUT) integer(i4b) On entry, degrees of freedom of the t-distribution. NDF must be greater or equal to 1.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probabilities to the right of T is calculated.
- UPPER = false : probabilities to the left of T is calculated.

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, if:

- NDF is lower or equal to NDF_MAX, the t_density is integrated.
- NDF is greater than NDF_MAX, an asymptotic series is used.

The default is 20.

Further Details

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-distribution (Algorithm 395). Comm. A.C.M., vol.13, 617-619

(2) **Cooper, B.E., 1968:** The integral of student's t distribution, (Algorithm AS3). Applied Statistics, vol.17, no.2, 189

6.15.40 function probt (t, ndf, upper, ndf_max)

Purpose

Evaluates the Student's t-distribution function from T(i,j) to infinity if UPPER is true or from minus infinity to T(i,j) if UPPER is false, for i=1 to size(T,1) and j=1 to size(T,2). In other words, if:

- UPPER = true : PROBT(i, j) = prob(U > T(i,j)) ,
- UPPER = false : PROBT(i, j) = prob(U < T(i,j)) ,

, for U = Student(NDF(i,j)), i=1 to size(T,1) and j=1 to size(T,2).

Arguments

T (INPUT) real(stnd), dimension(:,:) On entry, upper or lower limits of integration of the t-density.

NDF (INPUT) integer(i4b), dimension(:,:) On entry, degrees of freedom of the t-distribution. Any value in the array NDF must be greater or equal to 1.

The shape of NDF must verify:

- size(NDF,1) = size(T,1)
- size(NDF,2) = size(T,2).

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probabilities to the right of T is calculated.
- UPPER = false : probabilities to the left of T is calculated.

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, if:

- NDF is lower or equal to NDF_MAX, the t_density is integrated.
- NDF is greater than NDF_MAX, an asymptotic series is used.

The default is 20.

Further Details

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-distribution (Algorithm 395). Comm. A.C.M., vol.13, 617-619

(2) **Cooper, B.E., 1968:** The integral of student's t distribution, (Algorithm AS3). Applied Statistics, vol.17, no.2, 189

6.15.41 function probstudent (t, df)

Purpose

Evaluates the two-tailed probability of Student's t. PROBSTUDENT computes the probability that a random variable following Student's t distribution will exceed abs(T) in absolute value.

Arguments

T (INPUT) real(stnd) On entry, input constant. PROBSTUDENT computes the probability that abs(T) will be exceeded in absolute value.

DF (INPUT) real(stnd) On entry, degrees of freedom of the t-distribution. DF must be greater or equal to 1. DF is not necessarily an integer.

Further Details

This function is not very accurate for very small degrees of freedom (e.g. number of degrees of freedom less than 5).

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-distribution (Algorithm 395). Comm. A.C.M., vol.13, 617-619

6.15.42 function probstudent (t, df)

Purpose

Evaluates the two-tailed probabilities of Student's t. PROBSTUDENT computes probabilities that a random variable following Student's t distribution will exceed abs(T(i)) in absolute value, for i=1 to size(T).

Arguments

T (INPUT) real(stnd), dimension(:) On entry, input constants. PROBSTUDENT computes probabilities that abs(T(i)) will be exceeded in absolute value, for i=1 to size(T).

DF (INPUT) real(stnd) On entry, degrees of freedom of the t-distribution. DF must be greater or equal to 1. DF is not necessarily an integer.

Further Details

This function is not very accurate for very small degrees of freedom (e.g. number of degrees of freedom less than 5).

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-distribution (Algorithm 395). Comm. A.C.M., vol.13, 617-619

6.15.43 function probstudent (t, df)

Purpose

Evaluates the two-tailed probabilities of Student's t. PROBSTUDENT computes probabilities that a random variable following Student's t distribution will exceed abs(T(i)) in absolute value, for i=1 to size(T).

Arguments

T (INPUT) real(stnd), dimension(:) On entry, input constants. PROBSTUDENT computes probabilities that abs(T(i)) will be exceeded in absolute value, for i=1 to size(T).

DF (INPUT) real(stnd), dimension(:) On entry, degrees of freedom of the t-distribution. Any value in the array DF must be greater or equal to 1, but is not necessarily an integer.

The size of DF must be size(DF) = size(T) .

Further Details

This function is not very accurate for very small degrees of freedom (e.g. number of degrees of freedom less than 5).

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-distribution (Algorithm 395). Comm. A.C.M., vol.13, 617-619

6.15.44 function probstudent (t, df)

Purpose

Evaluates two-tailed probabilities of Student's t. PROBSTUDENT computes probabilities that a random variable following Student's t distribution will exceed abs(T(i,j)) in absolute value, for i=1 to size(T,1) and j=1 to size(T,2).

Arguments

T (INPUT) real(stnd), dimension(:,:) On entry, input constants. PROBSTUDENT computes probabilities that abs(T(i,j)) will be exceeded in absolute value, for i=1 to size(T,1) and j=1 to size(T,2).

DF (INPUT) real(stnd) On entry, degrees of freedom of the t-distribution. DF must be greater or equal to 1. DF is not necessarily an integer.

Further Details

This function is not very accurate for very small degrees of freedom (e.g. number of degrees of freedom less than 5).

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-distribution (Algorithm 395). Comm. A.C.M., vol.13, 617-619

6.15.45 function probstudent (t, df)

Purpose

Evaluates two-tailed probabilities of Student's t. PROBSTUDENT computes probabilities that a random variable following Student's t distribution will exceed abs(T(i,j)) in absolute value, for i=1 to size(T,1) and j=1 to size(T,2).

Arguments

T (INPUT) real(stnd), dimension(:,:) On entry, input constants. PROBSTUDENT computes probabilities that abs(T(i,j)) will be exceeded in absolute value, for i=1 to size(T,1) and j=1 to size(T,2).

DF (INPUT) real(stnd), dimension(:,:) On entry, degrees of freedom of the t-distribution. Any value in the array DF must be greater or equal to 1, but is not necessarily an integer.

The shape of DF must verify:

- size(DF,1) = size(T,1)
- size(DF,2) = size(T,2).

Further Details

This function is not very accurate for very small degrees of freedom (e.g. number of degrees of freedom less than 5).

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-distribution (Algorithm 395). Comm. A.C.M., vol.13, 617-619

6.15.46 function pinvt (p, ndf)

Purpose

Evaluates the inverse of the Student's t distribution function:

T0 = PINVT(P, NDF)

, if P = probability(U < T0) for U = Student(NDF).

PINVT returns the quantile T0 of Student's t-distribution with NDF degrees of freedom corresponding to a given lower tail area of P.

Arguments

P (INPUT) real(stnd) On entry, the probability. P must verify 0. < P < 1.

NDF (INPUT) integer(i4b) On entry, degrees of freedom of the t-distribution. NDF must be greater or equal to 1.

Further Details

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-quantiles (Algorithm 396). Comm. A.C.M., vol.13, no10, 620-621

6.15.47 function pinvt (p, ndf)

Purpose

Evaluates the inverse of the Student's t distribution function:

T0(i) = PINVT(P(i), NDF)

, if P(i) = probability(U < T0(i)) for U = Student(NDF) and i=1 to size(P).

PINVT returns the quantiles T0(i) of Student's t-distribution with NDF degrees of freedom corresponding to a given lower tail area of P(i), for i=1 to size(P).

Arguments

P (INPUT) real(stnd), dimension(:) On entry, the probabilities. P(i) must verify 0. < P(i) < 1, for i=1 to size(P).

NDF (INPUT) integer(i4b) On entry, degrees of freedom of the t-distribution. NDF must be greater or equal to 1.

Further Details

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-quantiles (Algorithm 396). Comm. A.C.M., vol.13, no10, 620-621

6.15.48 function pinvt (p, ndf)

Purpose

Evaluates the inverse of the Student's t distribution function:

T0(i) = PINVT(P(i), NDF(i))

, if P(i) = probability(U < T0(i)) for U = Student(NDF(i)) and i=1 to size(P).

PINVT returns the quantiles T0(i) of Student's t-distribution with NDF(i) degrees of freedom corresponding to a given lower tail area of P(i), for i=1 to size(P).

Arguments

P (INPUT) real(stnd), dimension(:) On entry, the probabilities. P(i) must verify 0. < P(i) < 1, for i=1 to size(P).

NDF (INPUT) integer(i4b), dimension(:) On entry, degrees of freedom of the t-distribution. Any value in the array NDF must be greater or equal to 1.

The size of NDF must be size(NDF) = size(P).

Further Details

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-quantiles (Algorithm 396). Comm. A.C.M., vol.13, no10, 620-621

6.15.49 function pinvt (p, ndf)

Purpose

Evaluates the inverse of the Student's t distribution function:

T0(i,j) = PINVT(P(i,j), NDF)

, if P(i,j) = probability(U < T0(i,j)) for U = Student(NDF), i=1 to size(P,1) and j=1 to size(P,2).

PINVT returns the quantiles T0(i,j) of Student's t-distribution with NDF degrees of freedom corresponding to a given lower tail area of P(i,j) for i=1 to size(P,1) and j=1 to size(P,2).

Arguments

P (INPUT) real(stnd), dimension(:,:) On entry, the probabilities. P(i,j) must verify 0. < P(i,j) < 1, for i=1 to size(P,1) and j=1 to size(P,2).

NDF (INPUT) integer(i4b) On entry, degrees of freedom of the t-distribution. NDF must be greater or equal to 1.

Further Details

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-quantiles (Algorithm 396). Comm. A.C.M., vol.13, no10, 620-621

6.15.50 function pinvt (p, ndf)

Purpose

Evaluates the inverse of the Student's t distribution function:

T0(i,j) = PINVT(P(i,j), NDF(i,j))

, if P(i,j) = probability(U < T0(i,j)) for U = Student(NDF(i,j)), i=1 to size(P,1) and j=1 to size(P,2).

PINVT returns the quantiles T0(i,j) of Student's t-distribution with NDF(i,j) degrees of freedom corresponding to a given lower tail area of P(i,j) for i=1 to size(P,1) and j=1 to size(P,2) .

Arguments

P (INPUT) real(stnd), dimension(:,:) On entry, the probabilities. P(i,j) must verify 0. < P(i,j) < 1, for i=1 to size(P,1) and j=1 to size(P,2).

NDF (INPUT) integer(i4b), dimension(:,:) On entry, degrees of freedom of the t-distribution. Any value in the array NDF must be greater or equal to 1.

The shape of NDF must verify:

- size(NDF,1) = size(P,1)
- size(NDF,2) = size(P,2).

Further Details

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-quantiles (Algorithm 396). Comm. A.C.M., vol.13, no10, 620-621

6.15.51 `function pinvstudent ( p, df )`

Purpose

Evaluates the inverse of a modification of Student's t probability distribution function.

PINVSTUDENT calculates the two-tail quantiles of Student's t-distribution, that is a value x such that the probability of the absolute value of t being greater than X is P.

Arguments

P (INPUT) real(stnd) On entry, the probability. P is the sum of the areas (equal) in both tails of the t-distribution. P must verify 0. < P < 1.

DF (INPUT) real(stnd) On entry, degrees of freedom of the t-distribution. DF must be greater or equal to 1. . DF is not necessarily an integer.

Further Details

Note that PINVSTUDENT does not provide the actual Student's t inverse. For q equal to the probability that a Student's t random variable is less than x, that inverse can be obtained by the following rules:

- for q in the range (0.0,0.5), call PINVSTUDENT with P = 2 * q and negate the result x.
- for q in the range (0.5,1.0), call PINVSTUDENT with P = 2 * (1-q).

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-quantiles (Algorithm 396). Comm. A.C.M., vol.13, no10, 620-621

6.15.52 function pinvstudent (p, df)

Purpose

Evaluates the inverse of a modification of Student's t probability distribution function.

PINVSTUDENT calculates the two-tail quantiles of Student's t-distribution, that is a value x(i) such that the probability of the absolute value of t being greater than x(i) is P(i), for i=1 to size(P).

Arguments

P (INPUT) real(stnd), dimension(:) On entry, the probabilities. P(i) is the sum of the areas (equal) in both tails of the t-distribution, for i=1 to size(P). P(i) must verify 0. < P(i) < 1, for i=1 to size(P).

DF (INPUT) real(stnd) On entry, degrees of freedom of the t-distribution. DF must be greater or equal to 1. . DF is not necessarily an integer.

Further Details

Note that PINVSTUDENT does not provide the actual Student's t inverse. For q(:) equal to the probabilities that a Student's t random variable is less than x(:), that inverse can be obtained by the following rules:

- for q(:) in the range (0.0,0.5), call PINVSTUDENT with P(:) = 2 * q(:) and negate the result x(:).
- for q(:) in the range (0.5,1.0), call PINVSTUDENT with P(:) = 2 * (1-q(:)).

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-quantiles (Algorithm 396). Comm. A.C.M., vol.13, no10, 620-621

6.15.53 function pinvstudent (p, df)

Purpose

Evaluates the inverse of a modification of Student's t probability distribution function.

PINVSTUDENT calculates the two-tail quantiles of Student's t-distribution, that is a value x(i) such that the probability of the absolute value of t being greater than x(i) is P(i), for i=1 to size(P).

Arguments

P (INPUT) real(stnd), dimension(:) On entry, the probabilities. P(i) is the sum of the areas (equal) in both tails of the t-distribution, for i=1 to size(P). P(i) must verify 0. < P(i) < 1, for i=1 to size(P).

DF (INPUT) real(stnd), dimension(:) On entry, degrees of freedom of the t-distribution. Any value in the array DF must be greater or equal to 1, but is not necessarily an integer.

The size of DF must be size(DF) = size(P) .

Further Details

Note that PINVSTUDENT does not provide the actual Student's t inverse. For q(:) equal to the probabilities that a Student's t random variable is less than x(:), that inverse can be obtained by the following rules:

- for q(:) in the range (0.0,0.5), call PINVSTUDENT with P(:) = 2 * q(:) and negate the result x(:).
- for q(:) in the range (0.5,1.0), call PINVSTUDENT with P(:) = 2 * (1-q(:)).

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-quantiles (Algorithm 396). Comm. A.C.M., vol.13, no10, 620-621

6.15.54 `function pinvstudent ( p, df )`

Purpose

Evaluates the inverse of a modification of Student's t probability distribution function.

PINVSTUDENT calculates the two-tail quantiles of Student's t-distribution, that is a value x(i,j) such that the probability of the absolute value of t being greater than x(i,j) is P(i,j), for i=1 to size(P,1) and j=1 to size(P,2).

Arguments

P (INPUT) real(stnd), dimension(:,:) On entry, the probabilities. P(i,j) is the sum of the areas (equal) in both tails of the t-distribution, for i=1 to size(P) and j=1 to size(P,2) . P(i,j) must verify 0. < P(i,j) < 1, for i=1 to size(P,1) and j=1 to size(P,2) .

DF (INPUT) real(stnd) On entry, degrees of freedom of the t-distribution. DF must be greater or equal to 1. . DF is not necessarily an integer.

Further Details

Note that PINVSTUDENT does not provide the actual Student's t inverse. For q(:,:) equal to the probabilities that a Student's t random variable is less than x(:,:), that inverse can be obtained by the following rules:

- for q(:,:) in the range (0.0,0.5), call PINVSTUDENT with P(:,:) = 2 * q(:,:) and negate the result x(:,:).
- for q(:,:) in the range (0.5,1.0), call PINVSTUDENT with P(:,:) = 2 * (1-q(:,:)).

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-quantiles (Algorithm 396). Comm. A.C.M., vol.13, no10, 620-621

6.15.55 function pinvstudent (p, df)

Purpose

Evaluates the inverse of a modification of Student's t probability distribution function.

PINVSTUDENT calculates the two-tail quantiles of Student's t-distribution, that is a value x(i,j) such that the probability of the absolute value of t being greater than x(i,j) is P(i,j), for i=1 to size(P,1) and j=1 to size(P,2).

Arguments

P (INPUT) real(stnd), dimension(:,:) On entry, the probabilities. P(i,j) is the sum of the areas (equal) in both tails of the t-distribution, for i=1 to size(P) and j=1 to size(P,2) . P(i,j) must verify 0. < P(i,j) < 1, for i=1 to size(P,1) and j=1 to size(P,2) .

DF (INPUT) real(stnd), dimension(:,:) On entry, degrees of freedom of the t-distribution. Any value in the array DF must be greater or equal to 1, but is not necessarily an integer.

The shape of DF must verify:

- size(DF,1) = size(P,1)
- size(DF,2) = size(P,2) .

Further Details

Note that PINVSTUDENT does not provide the actual Student's t inverse. For q(:,:) equal to the probabilities that a Student's t random variable is less than x(:,:), that inverse can be obtained by the following rules:

- for q(:,:) in the range (0.0,0.5), call PINVSTUDENT with P(:,:) = 2 * q(:,:) and negate the result x(:,:).
- for q(:,:) in the range (0.5,1.0), call PINVSTUDENT with P(:,:) = 2 * (1-q(:,:)).

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Hill, G.W., 1970:** Student's t-quantiles (Algorithm 396). Comm. A.C.M., vol.13, no10, 620-621

6.15.56 function probq (x2, ndf, upper, ndf_max)

Purpose

Evaluates the chi-squared distribution function from X2 to infinity if UPPER is true or from zero to X2 if UPPER is false. In other words, if:

- UPPER = true : PROBQ = prob(U > X2) ,
- UPPER = false : PROBQ = prob(U < X2) ,

, for U = Chi-squared(NDF).

Arguments

X2 (INPUT) real(stnd) On entry, upper or lower limit of integration. X2 must be greater or equal to zero.

NDF (INPUT) integer(i4b) On entry, degrees of freedom of the chi-squared distribution. NDF must be greater or equal to 1.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X2 is calculated.
- UPPER = false : probability to the left of X2 is calculated.

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, if:

- NDF is lower or equal to NDF_MAX, the chi-squared density is integrated.
- NDF is greater than NDF_MAX, a gaussian approximation is used.

The default is 40.

Further Details

If NDF<=NDF_MAX, the chi-squared distribution function is integrating by using formulae 26.4.4 and 26.4.5 in reference (1), otherwise a normal approximation based on the Wilson-Hilferty transformation is used (see the reference (1), Formula 26.4.14).

This function works only for integer degrees of freedom. It may be faster than PROBQ2 or PROBQ3 functions for the default value of NDF_MAX, but it is less accurate.

This function is adapted from:

(1) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 26.4.4, 26.4.5 and 26.4.14). New York, Dover Publications

(2) **Wilson, E.B., and Hilferty, M.M., 1931:** The distribution of Chi-square. Proceed. Nation. Academ. Scien., Vol. 17, 684-688

6.15.57 `function probq ( x2, ndf, upper, ndf_max )`

Purpose

Evaluates the chi-squared distribution function from X2(i) to infinity if UPPER is true or from zero to X2(i) if UPPER is false, for i=1 to size(X2). In other words, if:

- UPPER = true : PROBQ(i) = prob(U > X2(i)) ,
- UPPER = false : PROBQ(i) = prob(U < X2(i)) ,

, for U = Chi-squared(NDF) and i=1 to size(X2).

Arguments

X2 (INPUT) real(stnd), dimension(:) On entry, upper or lower limits of integration. X2(i) must be greater or equal to zero for i=1 to size(X2).

NDF (INPUT) integer(i4b) On entry, degrees of freedom of the chi-squared distribution. NDF must be greater or equal to 1.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X2 is calculated.
- UPPER = false : probability to the left of X2 is calculated.

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, if:

- NDF is lower or equal to NDF_MAX, the chi-squared density is integrated.
- NDF is greater than NDF_MAX, a gaussian approximation is used.

The default is 40.

Further Details

If NDF<=NDF_MAX, the chi-squared distribution function is integrating by using formulae 26.4.4 and 26.4.5 in reference (1), otherwise a normal approximation based on the Wilson-Hilferty transformation is used (see the reference (1), Formula 26.4.14).

This function works only for integer degrees of freedom. It may be faster than PROBQ2 or PROBQ3 functions for the default value of NDF_MAX, but it is less accurate.

The function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 26.4.4, 26.4.5 and 26.4.14). New York, Dover Publications

(2) **Wilson, E.B., and Hilferty, M.M., 1931:** The distribution of Chi-square. Proceed. Nation. Academ. Scien., Vol. 17, 684-688

6.15.58 `function probq ( x2, ndf, upper, ndf_max )`

Purpose

Evaluates the chi-squared distribution function from X2(i) to infinity if UPPER is true or from zero to X2(i) if UPPER is false, for i=1 to size(X2). In other words, if:

- UPPER = true : PROBQ(i) = prob(U > X2(i)) ,
- UPPER = false : PROBQ(i) = prob(U < X2(i)) ,

, for U = Chi-squared(NDF(i)) and i=1 to size(X2).

Arguments

X2 (INPUT) real(stnd), dimension(:) On entry, upper or lower limits of integration. X2(i) must be greater or equal to zero for i=1 to size(X2).

NDF (INPUT) integer(i4b), dimension(:) On entry, degrees of freedom of the chi-squared distribution. Any value in the array NDF must be greater or equal to 1.

The size of NDF must verify size(NDF) = size(X2) .

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X2 is calculated.
- UPPER = false : probability to the left of X2 is calculated.

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, if:

- NDF is lower or equal to NDF_MAX, the chi-squared density is integrated.
- NDF is greater than NDF_MAX, a gaussian approximation is used.

The default is 40.

Further Details

If NDF(i)<=NDF_MAX, the chi-squared distribution function is integrating by using formulae 26.4.4 and 26.4.5 in reference (1), otherwise a normal approximation based on the Wilson-Hilferty transformation is used (see the reference (1), Formula 26.4.14).

This function works only for integer degrees of freedom. It may be faster than PROBQ2 or PROBQ3 functions for the default value of NDF_MAX, but it is less accurate.

The function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 26.4.4, 26.4.5 and 26.4.14). New York, Dover Publications

(2) **Wilson, E.B., and Hilferty, M.M., 1931:** The distribution of Chi-square. Proceed. Nation. Academ. Scien., Vol. 17, 684-688

6.15.59 `function probq ( x2, ndf, upper, ndf_max )`

Purpose

Evaluates the chi-squared distribution function from X2(i,j) to infinity if UPPER is true or from zero to X2(i,j) if UPPER is false, for i=1 to size(X2,1) and j=1 to size(X2,2). In other words, if:

- UPPER = true : PROBQ(i, j) = prob(U > X2(i,j)) ,
- UPPER = false : PROBQ(i, j) = prob(U < X2(i,j)) ,

, for U = Chi-squared(NDF), i=1 to size(X2,1) and j=1 to size(X2,2).

Arguments

X2 (INPUT) real(stnd), dimension(:,:) On entry, upper or lower limits of integration. X2(i,j) must be greater or equal to zero for i=1 to size(X2,1) and j=1 to size(X2,2).

NDF (INPUT) integer(i4b) On entry, degrees of freedom of the chi-squared distribution. NDF must be greater or equal to 1.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X2 is calculated.
- UPPER = false : probability to the left of X2 is calculated.

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, if:

- NDF is lower or equal to NDF_MAX, the chi-squared density is integrated.
- NDF is greater than NDF_MAX, a gaussian approximation is used.

The default is 40.

Further Details

If NDF<=NDF_MAX, the chi-squared distribution function is integrating by using formulae 26.4.4 and 26.4.5 in reference (1), otherwise a normal approximation based on the Wilson-Hilferty transformation is used (see the reference (1), Formula 26.4.14).

This function works only for integer degrees of freedom. It may be faster than PROBQ2 or PROBQ3 functions for the default value of NDF_MAX, but it is less accurate.

The function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 26.4.4, 26.4.5 and 26.4.14). New York, Dover Publications

(2) **Wilson, E.B., and Hilferty, M.M., 1931:** The distribution of Chi-square. Proceed. Nation. Academ. Scien., Vol. 17, 684-688

6.15.60 `function probq ( x2, ndf, upper, ndf_max )`

Purpose

Evaluates the chi-squared distribution function from X2(i,j) to infinity if UPPER is true or from zero to X2(i,j) if UPPER is false, for i=1 to size(X2,1) and j=1 to size(X2,2). In other words, if:

- UPPER = true : PROBQ(i, j) = prob(U > X2(i,j)) ,
- UPPER = false : PROBQ(i, j) = prob(U < X2(i,j)) ,

, for U = Chi-squared(NDF(i, j)), i=1 to size(X2,1) and j=1 to size(X2,2).

Arguments

X2 (INPUT) real(stnd), dimension(:,:) On entry, upper or lower limits of integration. X2(i,j) must be greater or equal to zero for i=1 to size(X2,1) and j=1 to size(X2,2).

NDF (INPUT) integer(i4b), dimension(:,:) On entry, degrees of freedom of the chi-squared distribution. Any value in the array NDF must be greater or equal to 1.

The shape of NDF must verify:

- size(NDF,1) = size(X2,1)
- size(NDF,2) = size(X2,2) .

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X2 is calculated.
- UPPER = false : probability to the left of X2 is calculated.

NDF_MAX (INPUT, OPTIONAL) integer(i4b) On entry, if:

- NDF is lower or equal to NDF_MAX, the chi-squared density is integrated.
- NDF is greater than NDF_MAX, a gaussian approximation is used.

The default is 40.

Further Details

If NDF(i,j)<=NDF_MAX, the chi-squared distribution function is integrating by using formulae 26.4.4 and 26.4.5 in reference (1), otherwise a normal approximation based on the Wilson-Hilferty transformation is used (see the reference (1), Formula 26.4.14).

This function works only for integer degrees of freedom. It may be faster than PROBQ2 or PROBQ3 functions for the default value of NDF_MAX, but it is less accurate.

The function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 26.4.4, 26.4.5 and 26.4.14). New York, Dover Publications

(2) **Wilson, E.B., and Hilferty, M.M., 1931:** The distribution of Chi-square. Proceed. Nation. Academ. Scien., Vol. 17, 684-688

6.15.61 `function probq2 ( x2, df, upper, df_max, maxiter, failure )`

Purpose

Evaluates the chi-squared distribution function from X2 to infinity if UPPER is true or from zero to X2 if UPPER is false. In other words, if:

- UPPER = true : PROBQ2 = prob(U > X2) ,
- UPPER = false : PROBQ2 = prob(U <= X2) ,

, for U = Chi-squared(DF).

PROBQ2 computes the probability that a random variable which follows the chi-squared distribution with DF degrees of freedom is less than or equal to X2 (if UPPER is set to false) or greater than to X2 (if UPPER is set to true).

Arguments

X2 (INPUT) real(stnd) On entry, upper or lower limit of integration. X2 must be greater or equal to zero.

DF (INPUT) real(stnd) On entry, degrees of freedom of the chi-squared distribution. DF must be greater or equal to 0.5 and less than or equal to 200 000.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X2 is calculated.
- UPPER = false : probability to the left of X2 is calculated.

DF_MAX (INPUT, OPTIONAL) real(stnd) On entry, if:

- DF is lower or equal to DF_MAX, the chi-squared density is integrated using the incomplete Gamma integral.
- DF is greater than DF_MAX, a gaussian approximation is used.

The default is 100.

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series expansion of the incomplete Gamma integral if DF<=DF_MAX.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process of the incomplete Gamma integral did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

This argument is actually used only if DF<=DF_MAX.

The default value is false.

Further Details

If DF<=DF_MAX, the chi-squared distribution function is evaluated by integrating the incomplete Gamma integral (see the references (2) and (3) for more details), otherwise a normal approximation based on the Wilson-Hilferty transformation is used (see the reference (3), Formula 26.4.14, p.941).

This function is faster than PROBQ3 function, but is less accurate.

This function is adapted from:

(1) **Wilson, E.B., and Hilferty, M.M., 1931:** The distribution of Chi-square. Proceed. Nation. Academ. Scien., Vol. 17, 684-688

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-Squared and incomplete Gamma integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29, 6.5.32 and 26.4.14). New York, Dover Publications

6.15.62 function probq2 (x2, df, upper, df_max, maxiter, failure)

Purpose

Evaluates the chi-squared distribution function from X2(i) to infinity if UPPER is true or from zero to X2(i) if UPPER is false, for i=1 to size(X2). In other words, if:

- UPPER = true : PROBQ2(i) = prob(U > X2(i)) ,
- UPPER = false : PROBQ2(i) = prob(U <= X2(i)) ,

, for U = Chi-squared(DF) and i=1 to size(X2).

PROBQ2 computes the probabilities that a random variable (vector) which follows the chi-squared distribution with DF degrees of freedom is less than or equal to X2(:) (if UPPER is set to false) or greater than to X2(:) (if UPPER is set to true).

Arguments

X2 (INPUT) real(stnd), dimension(:) On entry, upper or lower limits of integration. X2(i) must be greater or equal to zero for i=1 to size(X2).

DF (INPUT) real(stnd) On entry, degrees of freedom of the chi-squared distribution. DF must be greater or equal to 0.5 and less than or equal to 200 000.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X2 is calculated.
- UPPER = false : probability to the left of X2 is calculated.

DF_MAX (INPUT, OPTIONAL) real(stnd) On entry, if:

- DF is lower or equal to DF_MAX, the chi-squared density is integrated using the incomplete Gamma integral.
- DF is greater than DF_MAX, a gaussian approximation is used.

The default is 100.

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series expansion of the incomplete Gamma integral if DF<=DF_MAX.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process of the incomplete Gamma integral did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

This argument is actually used only if DF<=DF_MAX.

The default value is false.

Further Details

If DF<=DF_MAX, the chi-squared distribution function is evaluated by integrating the incomplete Gamma integral (see the references (2) and (3) for more details), otherwise a normal approximation based on the Wilson-Hilferty transformation is used (see the reference (3), Formula 26.4.14, p.941).

This function is faster than PROBQ3 function, but is less accurate.

This functon is parallelized if OPENMP is used.

This function is adapted from:

(1) **Wilson, E.B., and Hilferty, M.M., 1931:** The distribution of Chi-square. Proceed. Nation. Academ. Scien., Vol. 17, 684-688

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-Squared and incomplete Gamma integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29, 6.5.32 and 26.4.14). New York, Dover Publications

6.15.63 function probq2 (x2, df, upper, df_max, maxiter, failure)

Purpose

Evaluates the chi-squared distribution function from X2(i) to infinity if UPPER is true or from zero to X2(i) if UPPER is false, for i=1 to size(X2). In other words, if:

- UPPER = true : PROBQ2(i) = prob(U > X2(i)) ,
- UPPER = false : PROBQ2(i) = prob(U <= X2(i)) ,

, for U = Chi-squared(DF(i)) and i=1 to size(X2).

PROBQ2 computes the probabilities that a random variable (vector) which follows the chi-squared distribution with DF(:) degrees of freedom is less than or equal to X2(:) (if UPPER is set to false) or greater than to X2(:) (if UPPER is set to true).

Arguments

X2 (INPUT) real(stnd), dimension(:) On entry, upper or lower limits of integration. X2(i) must be greater or equal to zero for i=1 to size(X2).

DF (INPUT) real(stnd), dimension(:) On entry, degrees of freedom of the chi-squared distribution. Any value in the array DF must be greater or equal to 0.5 and less than or equal to 200 000.

The size of DF must verify size(DF) = size(X2).

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X2 is calculated.
- UPPER = false : probability to the left of X2 is calculated.

DF_MAX (INPUT, OPTIONAL) real(stnd) On entry, if:

- DF is lower or equal to DF_MAX, the chi-squared density is integrated using the incomplete Gamma integral.
- DF is greater than DF_MAX, a gaussian approximation is used.

The default is 100.

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series expansion of the incomplete Gamma integral if DF<=DF_MAX.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process of the incomplete Gamma integral did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

This argument is actually used only for the values of DF(:) less than or equal to DF_MAX.

The default value is false.

Further Details

If DF(i)<=DF_MAX, the chi-squared distribution function is evaluated by integrating the incomplete Gamma integral (see the references (2) and (3) for more details), otherwise a normal approximation based on the Wilson-Hilferty transformation is used (see the reference (3), Formula 26.4.14, p.941).

This function is faster than PROBQ3 function, but is less accurate.

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Wilson, E.B., and Hilferty, M.M., 1931:** The distribution of Chi-square. Proceed. Nation. Academ. Scien., Vol. 17, 684-688

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-Squared and incomplete Gamma integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29, 6.5.32 and 26.4.14). New York, Dover Publications

6.15.64 function probq2 (x2, df, upper, df_max, maxiter, failure)

Purpose

Evaluates the chi-squared distribution function from X2(i,j) to infinity if UPPER is true or from zero to X2(i,j) if UPPER is false, for i=1 to size(X2,1) and j=1 to size(X2,2). In other words, if:

- UPPER = true : PROBQ2(i,j) = prob(U > X2(i,j)) ,
- UPPER = false : PROBQ2(i,j) = prob(U <= X2(i,j)) ,

, for U = Chi-squared(DF), i=1 to size(X2,1) and j=1 to size(X2,2).

PROBQ2 computes the probabilities that a random variable (matrix) which follows the chi-squared distribution with DF degrees of freedom is less than or equal to X2(:,:) (if UPPER is set to false) or greater than to X2(:,:) (if UPPER is set to true).

Arguments

X2 (INPUT) real(stnd), dimension(:,:) On entry, upper or lower limits of integration. X2(i,j) must be greater or equal to zero for i=1 to size(X2,1) and j=1 to size(X2,2).

DF (INPUT) real(stnd) On entry, degrees of freedom of the chi-squared distribution. DF must be greater or equal to 0.5 and less than or equal to 200 000.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X2 is calculated.
- UPPER = false : probability to the left of X2 is calculated.

DF_MAX (INPUT, OPTIONAL) real(stnd) On entry, if:

- DF is lower or equal to DF_MAX, the chi-squared density is integrated using the incomplete Gamma integral.
- DF is greater than DF_MAX, a gaussian approximation is used.

The default is 100.

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series expansion of the incomplete Gamma integral if DF<=DF_MAX.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process of the incomplete Gamma integral did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

This argument is actually used only if DF<=DF_MAX.

The default value is false.

Further Details

If DF<=DF_MAX, the chi-squared distribution function is evaluated by integrating the incomplete Gamma integral (see the references (2) and (3) for more details), otherwise a normal approximation based on the Wilson-Hilferty transformation is used (see the reference (3), Formula 26.4.14, p.941).

This function is faster than PROBQ3 function, but is less accurate.

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Wilson, E.B., and Hilferty, M.M., 1931:** The distribution of Chi-square. Proceed. Nation. Academ. Scien., Vol. 17, 684-688

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-Squared and incomplete Gamma integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29, 6.5.32 and 26.4.14). New York, Dover Publications

6.15.65 function probq2 (x2, df, upper, df_max, maxiter, failure)

Purpose

Evaluates the chi-squared distribution function from X2(i,j) to infinity if UPPER is true or from zero to X2(i,j) if UPPER is false, for i=1 to size(X2,1) and j=1 to size(X2,2). In other words, if:

- UPPER = true : PROBQ2(i,j) = prob(U > X2(i,j)) ,
- UPPER = false : PROBQ2(i,j) = prob(U <= X2(i,j)) ,

, for U = Chi-squared(DF(i,j)), i=1 to size(X2,1) and j=1 to size(X2,2).

PROBQ2 computes the probabilities that a random variable (matrix) which follows the chi-squared distribution with DF(:,:) degrees of freedom is less than or equal to X2(:,:) (if UPPER is set to false) or greater than to X2(:,:) (if UPPER is set to true).

Arguments

X2 (INPUT) real(stnd), dimension(:,:) On entry, upper or lower limits of integration. X2(i,j) must be greater or equal to zero for i=1 to size(X2,1) and j=1 to size(X2,2).

DF (INPUT) real(stnd), dimension(:,:) On entry, degrees of freedom of the chi-squared distribution. Any value in the array DF must be greater or equal to 0.5 and less than or equal to 200 000.

The shape of DF must verify:

- size(DF,1) = size(X2,1)
- size(DF,2) = size(X2,2).

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X2 is calculated.
- UPPER = false : probability to the left of X2 is calculated.

DF_MAX (INPUT, OPTIONAL) real(stnd) On entry, if:

- DF is lower or equal to DF_MAX, the chi-squared density is integrated using the incomplete Gamma integral.
- DF is greater than DF_MAX, a gaussian approximation is used.

The default is 100.

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series expansion of the incomplete Gamma integral if DF<=DF_MAX.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process of the incomplete Gamma integral did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

This argument is actually used only for the values of DF(:,:) less than or equal to DF_MAX.

The default value is false.

Further Details

If DF(i,j)<=DF_MAX, the chi-squared distribution function is evaluated by integrating the incomplete Gamma integral (see the references (2) and (3) for more details), otherwise a normal approximation based on the Wilson-Hilferty transformation is used (see the reference (3), Formula 26.4.14, p.941).

This function is faster than PROBQ3 function, but is less accurate.

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Wilson, E.B., and Hilferty, M.M., 1931:** The distribution of Chi-square. Proceed. Nation. Academ. Scien., Vol. 17, 684-688

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-Squared and incomplete Gamma integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29, 6.5.32 and 26.4.14). New York, Dover Publications

6.15.66 function probq3 (x2, df, upper, df_max, acu, maxiter, failure)

Purpose

Evaluates the chi-squared distribution function from X2 to infinity if UPPER is true or from zero to X2 if UPPER is false. In other words, if:

- UPPER = true : PROBQ3 = prob(U > X2) ,
- UPPER = false : PROBQ3 = prob(U <= X2) ,

, for U = Chi-squared(DF).

PROBQ3 computes the probability that a random variable which follows the chi-squared distribution with DF degrees of freedom is less than or equal to X2 (if UPPER is set to false) or greater than to X2 (if UPPER is set to true).

Arguments

X2 (INPUT) real(stnd) On entry, upper or lower limit of integration. X2 must be greater or equal to zero.

DF (INPUT) real(stnd) On entry, degrees of freedom of the chi-squared distribution. DF must be greater or equal to 0.5.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X2 is calculated.
- UPPER = false : probability to the left of X2 is calculated.

DF_MAX (INPUT, OPTIONAL) real(stnd) On entry, if:

- DF is lower or equal to DF_MAX, the chi-squared density is integrated using the incomplete Gamma integral.
- DF is greater than DF_MAX, a gaussian approximation is used.

The default is 100.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result if DF<=DF_MAX (e.g. if the incomplete Gamma integral is used). If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series or continued fraction expansion of the incomplete Gamma integral if DF<=DF_MAX.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process of the incomplete Gamma integral did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

This argument is actually used only if DF<=DF_MAX.

The default value is false.

Further Details

If DF<=DF_MAX, the chi-squared distribution function is evaluated by integrating the incomplete Gamma integral (see the references (2) and (3) for more details), otherwise a normal approximation based on the Wilson-Hilferty transformation is used (see the reference (3), Formula 26.4.14, p.941).

This function is more accurate than PROBQ and PROBQ2 functions, but it is slower.

This function is adapted from:

(1) **Wilson, E.B., and Hilferty, M.M., 1931:** The distribution of Chi-square. Proceed. Nation. Academ. Scien., Vol. 17, 684-688

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-Squared and incomplete Gamma integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29, 6.5.31 and 26.4.14). New York, Dover Publications

6.15.67 function probq3 (x2, df, upper, df_max, acu, maxiter, failure)

Purpose

Evaluates the chi-squared distribution function from X2(i) to infinity if UPPER is true or from zero to X2(i) if UPPER is false, for i=1 to size(X2). In other words, if:

- UPPER = true : PROBQ3(i) = prob(U > X2(i)) ,
- UPPER = false : PROBQ3(i) = prob(U <= X2(i)) ,

, for U = Chi-squared(DF) and i=1 to size(X2).

PROBQ3 computes the probabilities that a random variable (vector) which follows the chi-squared distribution with DF degrees of freedom is less than or equal to X2(:) (if UPPER is set to false) or greater than to X2(:) (if UPPER is set to true).

Arguments

X2 (INPUT) real(stnd), dimension(:) On entry, upper or lower limits of integration. X2(i) must be greater or equal to zero for i=1 to size(X2).

DF (INPUT) real(stnd) On entry, degrees of freedom of the chi-squared distribution. DF must be greater or equal to 0.5.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X2 is calculated.
- UPPER = false : probability to the left of X2 is calculated.

DF_MAX (INPUT, OPTIONAL) real(stnd) On entry, if:

- DF is lower or equal to DF_MAX, the chi-squared density is integrated using the incomplete Gamma integral.
- DF is greater than DF_MAX, a gaussian approximation is used.

The default is 100.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result if DF<=DF_MAX (e.g. if the incomplete Gamma integral is used). If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series or continued fraction expansion of the incomplete Gamma integral if DF<=DF_MAX.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process of the incomplete Gamma integral did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

This argument is actually used only if DF<=DF_MAX.

The default value is false.

Further Details

If DF<=DF_MAX, the chi-squared distribution function is evaluated by integrating the incomplete Gamma integral (see the references (2) and (3) for more details), otherwise a normal approximation based on the Wilson-Hilferty transformation is used (see the reference (3), Formula 26.4.14, p.941).

This function is more accurate than PROBQ and PROBQ2 functions, but it is slower.

The function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Wilson, E.B., and Hilferty, M.M., 1931:** The distribution of Chi-square. Proceed. Nation. Academ. Scien., Vol. 17, 684-688

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-Squared and incomplete Gamma integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29, 6.5.31 and 26.4.14). New York, Dover Publications

6.15.68 function probq3 (x2, df, upper, df_max, acu, maxiter, failure)

Purpose

Evaluates the chi-squared distribution function from X2(i) to infinity if UPPER is true or from zero to X2(i) if UPPER is false, for i=1 to size(X2). In other words, if:

- UPPER = true : PROBQ3(i) = prob(U > X2(i)) ,
- UPPER = false : PROBQ3(i) = prob(U <= X2(i)) ,

, for U = Chi-squared(NDF(i)) and i=1 to size(X2).

PROBQ3 computes the probabilities that a random variable (vector) which follows the chi-squared distribution with DF(:) degrees of freedom is less than or equal to X2(:) (if UPPER is set to false) or greater than to X2(:) (if UPPER is set to true).

Arguments

X2 (INPUT) real(stnd), dimension(:) On entry, upper or lower limits of integration. X2(i) must be greater or equal to zero for i=1 to size(X2).

DF (INPUT) real(stnd), dimension(:) On entry, degrees of freedom of the chi-squared distribution. Any value in the array DF must be greater or equal to 0.5.

The size of NDF must verify size(DF) = size(X2) .

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X2 is calculated.
- UPPER = false : probability to the left of X2 is calculated.

DF_MAX (INPUT, OPTIONAL) real(stnd) On entry, if:

- DF is lower or equal to DF_MAX, the chi-squared density is integrated using the incomplete Gamma integral.
- DF is greater than DF_MAX, a gaussian approximation is used.

The default is 100.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result if DF<=DF_MAX (e.g. if the incomplete Gamma integral is used). If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series or continued fraction expansion of the incomplete Gamma integral if DF<=DF_MAX.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process of the incomplete Gamma integral did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

This argument is actually used only if DF<=DF_MAX.

The default value is false.

Further Details

If DF(i)<=DF_MAX, the chi-squared distribution function is evaluated by integrating the incomplete Gamma integral (see the references (2) and (3) for more details), otherwise a normal approximation based on the Wilson-Hilferty transformation is used (see the reference (3), Formula 26.4.14, p.941).

This function is more accurate than PROBQ and PROBQ2 functions, but it is slower.

The function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Wilson, E.B., and Hilferty, M.M., 1931:** The distribution of Chi-square. Proceed. Nation. Academ. Scien., Vol. 17, 684-688

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-Squared and incomplete Gamma integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29, 6.5.31 and 26.4.14). New York, Dover Publications

6.15.69 function probq3 (x2, df, upper, df_max, acu, maxiter, failure)

Purpose

Evaluates the chi-squared distribution function from X2(i,j) to infinity if UPPER is true or from zero to X2(i,j) if UPPER is false, for i=1 to size(X2,1) and j=1 to size(X2,2). In other words, if:

- UPPER = true : PROBQ3(i,j) = prob(U > X2(i,j)) ,
- UPPER = false : PROBQ3(i,j) = prob(U <= X2(i,j)) ,

, for U = Chi-squared(DF), i=1 to size(X2,1) and j=1 to size(X2,2).

PROBQ3 computes the probabilities that a random variable (matrix) which follows the chi-squared distribution with DF degrees of freedom is less than or equal to X2(:,:) (if UPPER is set to false) or greater than to X2(:,:) (if UPPER is set to true).

Arguments

X2 (INPUT) real(stnd), dimension(:,:) On entry, upper or lower limits of integration. X2(i,j) must be greater or equal to zero for i=1 to size(X2,1) and j=1 to size(X2,2).

DF (INPUT) real(stnd) On entry, degrees of freedom of the chi-squared distribution. DF must be greater or equal to 0.5.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X2 is calculated.
- UPPER = false : probability to the left of X2 is calculated.

DF_MAX (INPUT, OPTIONAL) real(stnd) On entry, if:

- DF is lower or equal to DF_MAX, the chi-squared density is integrated using the incomplete Gamma integral.
- DF is greater than DF_MAX, a gaussian approximation is used.

The default is 100.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result if DF<=DF_MAX (e.g. if the incomplete Gamma integral is used). If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series or continued fraction expansion of the incomplete Gamma integral if DF<=DF_MAX.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process of the incomplete Gamma integral did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

This argument is actually used only if DF<=DF_MAX.

The default value is false.

Further Details

If DF<=DF_MAX, the chi-squared distribution function is evaluated by integrating the incomplete Gamma integral (see the references (2) and (3) for more details), otherwise a normal approximation based on the Wilson-Hilferty transformation is used (see the reference (3), Formula 26.4.14, p.941).

This function is more accurate than PROBQ and PROBQ2 functions, but it is slower.

The function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Wilson, E.B., and Hilferty, M.M., 1931:** The distribution of Chi-square. Proceed. Nation. Academ. Scien., Vol. 17, 684-688

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-Squared and incomplete Gamma integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29, 6.5.31 and 26.4.14). New York, Dover Publications

6.15.70 function probq3 (x2, df, upper, df_max, acu, maxiter, failure)

Purpose

Evaluates the chi-squared distribution function from X2(i,j) to infinity if UPPER is true or from zero to X2(i,j) if UPPER is false, for i=1 to size(X2,1) and j=1 to size(X2,2). In other words, if:

- UPPER = true : PROBQ3(i, j) = prob(U > X2(i,j)) ,
- UPPER = false : PROBQ3(i, j) = prob(U < X2(i,j)) ,

, for U = Chi-squared(NDF(i, j)), i=1 to size(X2,1) and j=1 to size(X2,2).

PROBQ3 computes the probabilities that a random variable (matrix) which follows the chi-squared distribution with DF(:,:) degrees of freedom is less than or equal to X2(:,:) (if UPPER is set to false) or greater than to X2(:,:) (if UPPER is set to true).

Arguments

X2 (INPUT) real(stnd), dimension(:,:) On entry, upper or lower limits of integration. X2(i,j) must be greater or equal to zero for i=1 to size(X2,1) and j=1 to size(X2,2).

DF (INPUT) real(stnd), dimension(:,:) On entry, degrees of freedom of the chi-squared distribution. Any value in the array DF must be greater or equal to 0.5.

The shape of DF must verify:

- size(DF,1) = size(X2,1)
- size(DF,2) = size(X2,2) .

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of X2 is calculated.
- UPPER = false : probability to the left of X2 is calculated.

DF_MAX (INPUT, OPTIONAL) real(stnd) On entry, if:

- DF is lower or equal to DF_MAX, the chi-squared density is integrated using the incomplete Gamma integral.
- DF is greater than DF_MAX, a gaussian approximation is used.

The default is 100.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result if DF<=DF_MAX (e.g. if the incomplete Gamma integral is used). If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series or continued fraction expansion of the incomplete Gamma integral if DF<=DF_MAX.

The default value is 1000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process of the incomplete Gamma integral did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

This argument is actually used only if DF<=DF_MAX.

The default value is false.

Further Details

If DF(i,j)<=DF_MAX, the chi-squared distribution function is evaluated by integrating the incomplete Gamma integral (see the references (2) and (3) for more details), otherwise a normal approximation based on the Wilson-Hilferty transformation is used (see the reference (3), Formula 26.4.14, p.941).

This function is more accurate than PROBQ and PROBQ2 functions, but it is slower.

The function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Wilson, E.B., and Hilferty, M.M., 1931:** The distribution of Chi-square. Proceed. Nation. Academ. Scien., Vol. 17, 684-688

(2) **Shea, B.L., 1988:** Algorithm AS 239: Chi-Squared and incomplete Gamma integral. Appl. Statist., Vol. 37, No. 3, pp. 466-473

(3) **Abramowitz, M., and Stegun, I.A., 1970:** Handbook of Mathematical Functions, (formulae 6.5.29, 6.5.31 and 26.4.14). New York, Dover Publications

6.15.71 `function pinvq ( p, ndf )`

Purpose

Evaluates the inverse of the chi-squared distribution function:

X0 = PINVQ(P, NDF)

, if P = probability(U < X0) for U = Chi-squared(NDF).

PINVQ returns the quantile X0 of the chi-squared distribution with NDF degrees of freedom corresponding to a given lower tail area of P.

Arguments

P (INPUT) real(stnd) On entry, the probability. P must verify 0. < P < 1. .

NDF (INPUT) integer(i4b) On entry, degrees of freedom of the chi-squared distribution. NDF must be greater or equal to 1.

Further Details

This function is fast, but not very accurate especially for small degrees of freedom, e.g. for NDF<10 or 20. If high accuracy is desired, function PINVQ2 must be used instead.

This function is adapted from:

(1) **Goldstein, R.B., 1973:** Chi-square quantiles. Comm. A.C.M., vol.16, no.8, 483-485

6.15.72 function pinvq (p, ndf)

Purpose

Evaluates the inverse of the chi-squared distribution function:

X0(i) = PINVQ(P(i), NDF)

, if P(i) = probability(U < X0(i)) for U = Chi-squared(NDF) and i=1 to size(P).

PINVQ returns the quantiles X0(i) of the chi-squared distribution with NDF degrees of freedom corresponding to a given lower tail area of P(i), for i=1 to size(P).

Arguments

P (INPUT) real(stnd), dimension(:) On entry, the probabilities. P(i) must verify 0. < P(i) < 1. , for i=1 to size(P).

NDF (INPUT) integer(i4b) On entry, degrees of freedom of the chi-squared distribution. NDF must be greater or equal to 1.

Further Details

This function is fast, but not very accurate especially for small degrees of freedom, e.g. for NDF<10 or 20. If high accuracy is desired, function PINVQ2 must be used instead.

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Goldstein, R.B., 1973:** Chi-square quantiles. Comm. A.C.M., vol.16, no.8, 483-485

6.15.73 function pinvq (p, ndf)

Purpose

Evaluates the inverse of the chi-squared distribution function:

X0(i) = PINVQ(P(i), NDF)

, if P(i) = probability(U < X0(i)) for U = Chi-squared(NDF(i)) and i=1 to size(P).

PINVQ returns the quantiles X0(i) of the chi-squared distribution with NDF(i) degrees of freedom corresponding to a given lower tail area of P(i), for i=1 to size(P).

Arguments

P (INPUT) real(stnd), dimension(:) On entry, the probabilities. P(i) must verify 0. < P(i) < 1. , for i=1 to size(P).

NDF (INPUT) integer(i4b), dimension(:) On entry, degrees of freedom of the chi-squared distribution. Any value in the array NDF must be greater or equal to 1.

The size of NDF must be size(NDF) = size(P).

Further Details

This function is fast, but not very accurate especially for small degrees of freedom, e.g. for NDF<10 or 20. If high accuracy is desired, function PINVQ2 must be used instead.

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Goldstein, R.B., 1973:** Chi-square quantiles. Comm. A.C.M., vol.16, no.8, 483-485

6.15.74 function pinvq (p, ndf)

Purpose

Evaluates the inverse of the chi-squared distribution function:

X0(i,j) = PINVQ(P(i,j), NDF)

, if P(i,j) = probability(U < T0(i,j)) for U = Chi-squared(NDF), i=1 to size(P,1) and j=1 to size(P,2).

PINVQ returns the quantiles X0(i,j) of the chi-squared distribution with NDF degrees of freedom corresponding to a given lower tail area of P(i,j) for i=1 to size(P,1) and j=1 to size(P,2).

Arguments

P (INPUT) real(stnd), dimension(:,:) On entry, the probabilities. P(i,j) must verify 0. < P(i,j) < 1. , for i=1 to size(P,1) and j=1 to size(P,2).

NDF (INPUT) integer(i4b) On entry, degrees of freedom of the chi-squared distribution. NDF must be greater or equal to 1.

Further Details

This function is fast, but not very accurate especially for small degrees of freedom, e.g. for NDF<10 or 20. If high accuracy is desired, function PINVQ2 must be used instead.

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Goldstein, R.B., 1973:** Chi-square quantiles. Comm. A.C.M., vol.16, no.8, 483-485

6.15.75 function pinvq (p, ndf)

Purpose

Evaluates the inverse of the chi-squared distribution function:

X0(i,j) = PINVQ(P(i,j), NDF)

, if P(i,j) = probability(U < T0(i,j)) for U = Chi-squared(NDF(i,j)), i=1 to size(P,1) and j=1 to size(P,2).

PINVQ returns the quantiles X0(i,j) of the chi-squared distribution with NDF(i,j) degrees of freedom corresponding to a given lower tail area of P(i,j) for i=1 to size(P,1) and j=1 to size(P,2).

Arguments

P (INPUT) real(stnd), dimension(:,:) On entry, the probabilities. P(i,j) must verify 0. < P(i,j) < 1. , for i=1 to size(P,1) and j=1 to size(P,2).

NDF (INPUT) integer(i4b), dimension(:,:) On entry, degrees of freedom of the chi-squared distribution. Any value in the array NDF must be greater or equal to 1.

The shape of NDF must verify:

- size(NDF,1) = size(P,1)
- size(NDF,2) = size(P,2).

Further Details

This function is fast, but not very accurate especially for small degrees of freedom, e.g. for NDF<10 or 20. If high accuracy is desired, function PINVQ2 must be used instead.

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Goldstein, R.B., 1973:** Chi-square quantiles. Comm. A.C.M., vol.16, no.8, 483-485

6.15.76 function pinvq2 (p, df, prec, acu, maxiter)

Purpose

Evaluates the inverse of the chi-squared distribution function:

X0 = PINVQ2(P, DF)

, if P = probability(U < X0) for U = Chi-squared(DF).

PINVQ2 returns the quantile X0 of the chi-squared distribution with DF degrees of freedom corresponding to a given lower tail area of P. In other words, PINVQ2 outputs a chi-squared value, X0, such that a random variable, distributed as chi-squared with DF degrees of freedom, will be less than or equal to X0 with probability P.

Arguments

P (INPUT) real(stnd) On entry, the probability. P must be in the inclusive range (0,1).

DF (INPUT) real(stnd) On entry, degrees of freedom of the chi-squared distribution. DF must be greater or equal to 0.5.

PREC (INPUT, OPTIONAL) real(stnd) On entry, the desired accurary of the result. If more than six significant digits are required, the default value of PREC (e.g. 0.5e-06_stnd) should be altered appropriately (e.g. decreased). PREC is a small strictly positive integer less than 0.5e-06_stnd.

The default value for PREC is 0.5e-06_stnd .

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result when computing the incomplete Gamma integral in the evaluation of the seven term Taylor series. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series or continued fraction expansion of the incomplete Gamma integral.

See the description of the PROBGAMMA2 function for more details on this argument.

The default value is 1000.

Further Details

This function is both more general (here the number of degrees of freedom, DF, is not necessarily an integer) and more accurate (here the quantile X0 may be calculated as exactly as the computer allows with the parameter PREC) than PINVQ function.

This function is adapted from:

(1) **Best, D.J., and Roberts, D.E., 1975:** Algorithm AS 91: The Percentage Points of the chi2 Distribution. Appl. Statist., Vol.24, No. 3, pp.385-388

(2) **Shea, B.L., 1991:** Algorithm AS R85: A remark on AS 91: The Percentage Points of the chi2 Distribution. Appl. Statist. Vol.40, No. 1, pp.233-235.

6.15.77 `function pinvq2 ( p, df, prec, acu, maxiter )`

Purpose

Evaluates the inverse of the chi-squared distribution function:

X0 = PINVQ2(P(i), DF)

, if P(i) = probability(U < X0) for U = Chi-squared(DF) and i=1 to size(P).

PINVQ2 returns the quantiles X0(i) of the chi-squared distribution with DF degrees of freedom corresponding to a given lower tail area of P(i), for i=1 to size(P). In other words, PINVQ2 outputs chi-squared values, X0(:), such that random variables, distributed as chi-squared with DF degrees of freedom, will be less than or equal to X0(:) with associated probabilities P(:).

Arguments

P (INPUT) real(stnd), dimension(:) On entry, the probabilities. P(i) must verify 0. < P(i) < 1, for i=1 to size(P).

DF (INPUT) real(stnd) On entry, degrees of freedom of the chi-squared distribution. DF must be greater or equal to 0.5.

PREC (INPUT, OPTIONAL) real(stnd) On entry, the desired accurary of the result. If more than six significant digits are required, the default value of PREC (e.g. 0.5e-06_stnd) should be altered appropriately(e.g. decreased). PREC is a small strictly positive integer less than 0.5e-06_stnd.

The default value for PREC is 0.5e-06_stnd .

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result when computing the incomplete Gamma integral in the evaluation of the seven term Taylor series. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series or continued fraction expansion of the incomplete Gamma integral.

See the description of the PROBGAMMA2 function for more details on this argument.

The default value is 1000.

Further Details

This function is both more general (here the number of degrees of freedom, DF, is not necessarily an integer) and more accurate (here the quantiles X0(:) may be calculated as exactly as the computer allows with the parameter PREC) than PINVQ function.

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Best, D.J., and Roberts, D.E., 1975:** Algorithm AS 91: The Percentage Points of the chi2 Distribution. Appl. Statist., Vol.24, No. 3, pp.385-388

(2) **Shea, B.L., 1991:** Algorithm AS R85: A remark on AS 91: The Percentage Points of the chi2 Distribution. Appl. Statist. Vol.40, No. 1, pp.233-235.

6.15.78 `function pinvq2 ( p, df, prec, acu, maxiter )`

Purpose

Evaluates the inverse of the chi-squared distribution function:

X0 = PINVQ2(P(i), DF(i))

, if P(i) = probability(U < X0) for U = Chi-squared(DF(i)) and i=1 to size(P).

PINVQ2 returns the quantiles X0(i) of the chi-squared distribution with DF(i) degrees of freedom corresponding to a given lower tail area of P(i), for i=1 to size(P). In other words, PINVQ2 outputs chi-squared values, X0(:), such that random variables, distributed as chi-squared with DF(:) degrees of freedom, will be less than or equal to X0(:) with associated probabilities P(:).

Arguments

P (INPUT) real(stnd), dimension(:) On entry, the probabilities. P(i) must verify 0. < P(i) < 1, for i=1 to size(P) .

DF (INPUT) real(stnd), dimension(:) On entry, degrees of freedom of the chi-squared distribution. Any value in the array DF must be greater or equal to 0.5.

The size of DF must verify size(DF) = size(P).

PREC (INPUT, OPTIONAL) real(stnd) On entry, the desired accurary of the result. If more than six significant digits are required, the default value of PREC (e.g. 0.5e-06_stnd) should be altered appropriately(e.g. decreased). PREC is a small strictly positive integer less than 0.5e-06_stnd.

The default value for PREC is 0.5e-06_stnd .

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result when computing the incomplete Gamma integral in the evaluation of the seven term Taylor series. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series or continued fraction expansion of the incomplete Gamma integral.

See the description of the PROBGAMMA2 function for more details on this argument.

The default value is 1000.

Further Details

This function is both more general (here the numbers of degrees of freedom, DF(:), are not necessarily integers) and more accurate (here the quantiles X0(:) may be calculated as exactly as the computer allows with the parameter PREC) than PINVQ function.

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Best, D.J., and Roberts, D.E., 1975:** Algorithm AS 91: The Percentage Points of the chi2 Distribution. Appl. Statist., Vol.24, No. 3, pp.385-388

(2) **Shea, B.L., 1991:** Algorithm AS R85: A remark on AS 91: The Percentage Points of the chi2 Distribution. Appl. Statist. Vol.40, No. 1, pp.233-235.

6.15.79 `function pinvq2 ( p, df, prec, acu, maxiter )`

Purpose

Evaluates the inverse of the chi-squared distribution function:

X0 = PINVQ2(P(i,j), DF)

, if P(i,j) = probability(U < X0) for U = Chi-squared(DF), i=1 to size(P,1) and j=1 to size(P,2).

PINVQ2 returns the quantiles X0(i,j) of the chi-squared distribution with DF degrees of freedom corresponding to a given lower tail area of P(i,j), for i=1 to size(P) and j=1 to size(P,2). In other words,

PINVQ2 outputs chi-squared values, X0(:,:), such that random variables, distributed as chi-squared with DF degrees of freedom, will be less than or equal to X0(:,:) with associated probabilities P(:,:).

Arguments

P (INPUT) real(stnd), dimension(:,:) On entry, the probabilities. P(i,j) must verify 0. < P(i,j) < 1, for i=1 to size(P) and j=1 to size(P,2).

DF (INPUT) real(stnd) On entry, degrees of freedom of the chi-squared distribution. DF must be greater or equal to 0.5.

PREC (INPUT, OPTIONAL) real(stnd) On entry, the desired accurary of the result. If more than six significant digits are required, the default value of PREC (e.g. 0.5e-06_stnd) should be altered appropriately(e.g. decreased). PREC is a small strictly positive integer less than 0.5e-06_stnd.

The default value for PREC is 0.5e-06_stnd .

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result when computing the incomplete Gamma integral in the evaluation of the seven term Taylor series. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series or continued fraction expansion of the incomplete Gamma integral.

See the description of the PROBGAMMA2 function for more details on this argument.

The default value is 1000.

Further Details

This function is both more general (here the number of degrees of freedom, DF, is not necessarily an integer) and more accurate (here the quantiles X0(:,:) may be calculated as exactly as the computer allows with the parameter PREC) than PINVQ function.

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Best, D.J., and Roberts, D.E., 1975:** Algorithm AS 91: The Percentage Points of the chi2 Distribution. Appl. Statist., Vol.24, No. 3, pp.385-388

(2) **Shea, B.L., 1991:** Algorithm AS R85: A remark on AS 91: The Percentage Points of the chi2 Distribution. Appl. Statist. Vol.40, No. 1, pp.233-235.

6.15.80 `function pinvq2 ( p, df, prec, acu, maxiter )`

Purpose

Evaluates the inverse of the chi-squared distribution function:

X0 = PINVQ2(P(i,j), DF(i,j))

, if P(i,j) = probability(U < X0) for U = Chi-squared(DF(i,j)), i=1 to size(P,1) and j=1 to size(P,2).

PINVQ2 returns the quantiles X0(i,j) of the chi-squared distribution with DF(i,j) degrees of freedom corresponding to a given lower tail area of P(i,j), for i=1 to size(P) and j=1 to size(P,2). In other words, PINVQ2 outputs chi-squared values, X0(:,:), such that random variables, distributed as chi-squared with corresponding DF(:,:) degrees of freedom, will be less than or equal to X0(:,:) with associated probabilities P(:,:).

Arguments

P (INPUT) real(stnd), dimension(:,:) On entry, the probabilities. P(i,j) must verify 0. < P(i,j) < 1, for i=1 to size(P) and j=1 to size(P,2).

DF (INPUT) real(stnd), dimension(:,:) On entry, degrees of freedom of the chi-squared distribution. Any value in the array DF must be greater or equal to 0.5.

The shape of DF must verify:

- size(DF,1) = size(P,1)
- size(DF,2) = size(P,2).

PREC (INPUT, OPTIONAL) real(stnd) On entry, the desired accurary of the result. If more than six significant digits are required, the default value of PREC (e.g. 0.5e-06_stnd) should be altered appropriately(e.g. decreased). PREC is a small strictly positive integer less than 0.5e-06_stnd.

The default value for PREC is 0.5e-06_stnd .

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result when computing the incomplete Gamma integral in the evaluation of the seven term Taylor series. If l decimal places of accuracy are required then ACU should be set to 10**(-(l+1)). ACU is a small strictly positive integer. ACU should not be set smaller than the machine precision since the stated accuracy cannot be attained. In that case the machine precision is used instead.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the Pearson's series or continued fraction expansion of the incomplete Gamma integral.

See the description of the PROBGAMMA2 function for more details on this argument.

The default value is 1000.

Further Details

This function is both more general (here the numbers of degrees of freedom, DF(:,:), are not necessarily integers) and more accurate (here the quantiles X0(:,:) may be calculated as exactly as the computer allows with the parameter PREC) than PINVQ function.

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Best, D.J., and Roberts, D.E., 1975:** Algorithm AS 91: The Percentage Points of the chi2 Distribution. Appl. Statist., Vol.24, No. 3, pp.385-388

(2) **Shea, B.L., 1991:** Algorithm AS R85: A remark on AS 91: The Percentage Points of the chi2 Distribution. Appl. Statist. Vol.40, No. 1, pp.233-235.

6.15.81 function probf (f, ndf1, ndf2, upper)

Purpose

Evaluates the F distribution function with degrees of freedom NDF1 and NDF2 from F to infinity if UPPER is true or from zero to F if UPPER is false. In other words, if:

- UPPER = true : PROBF = prob(U > F) ,
- UPPER = false : PROBF = prob(U < F) ,

, for U = Fisher(NDF1,NDF2).

Arguments

F (INPUT) real(stnd) On entry, upper or lower limit of integration. F must be greater or equal to zero.

NDF1 (INPUT) integer(i4b) On entry, first degree of freedom of the F distribution (numerator). NDF1 must be greater or equal to 1.

NDF2 (INPUT) integer(i4b) On entry, second degree of freedom of the F distribution (denominator). NDF2 must be greater or equal to 1.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of F is calculated.
- UPPER = false : probability to the left of F is calculated.

Further Details

This function accepts only integer values of degree of freedom and uses a normal approximation. This normal approximation is not accurate for small values of degrees of freedom.

This function is adapted from:

(1) **Peizer, D.B., and Pratt, J.W., 1968:** A normal approximation for Binomial, F, Beta, and . . . , (formula 2.24a). J.A.S.A., Vol. 63, 1457-1483

6.15.82 function probf (f, ndf1, ndf2, upper)

Purpose

Evaluates the F distribution function with degrees of freedom NDF1 and NDF2 from F(i) to infinity if UPPER is true or from zero to F(i) if UPPER is false, for i=1 to size(F). In other words, if:

- UPPER = true : PROBF(i) = prob(U > F(i)) ,
- UPPER = false : PROBF(i) = prob(U < F(i)) ,

, for U = Fisher(NDF1,NDF2) and i=1 to size(F).

Arguments

F (INPUT) real(stnd), dimension(:) On entry, upper or lower limits of integration. F(i) must be greater or equal to zero for i=1 to size(F).

NDF1 (INPUT) integer(i4b) On entry, first degree of freedom of the F distribution (numerator). NDF1 must be greater or equal to 1.

NDF2 (INPUT) integer(i4b) On entry, second degree of freedom of the F distribution (denominator). NDF2 must be greater or equal to 1.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of F is calculated.
- UPPER = false : probability to the left of F is calculated.

Further Details

This function accepts only integer values of degree of freedom and uses a normal approximation. This normal approximation is not accurate for small values of degrees of freedom.

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Peizer, D.B., and Pratt, J.W., 1968:** A normal approximation for Binomial, F, Beta, and . . . , (formula 2.24a). J.A.S.A., Vol. 63, 1457-1483

6.15.83 `function probf ( f, ndf1, ndf2, upper )`

Purpose

Evaluates the F distribution function with degrees of freedom NDF1 and NDF2 from F(i) to infinity if UPPER is true or from zero to F(i) if UPPER is false, for i=1 to size(F). In other words, if:

- UPPER = true : PROBF(i) = prob(U > F(i)) ,
- UPPER = false : PROBF(i) = prob(U < F(i)) ,

, for U = Fisher(NDF1(i),NDF2(i)) and i=1 to size(F).

Arguments

F (INPUT) real(stnd), dimension(:) On entry, upper or lower limits of integration. F(i) must be greater or equal to zero for i=1 to size(F).

NDF1 (INPUT) integer(i4b), dimension(:) On entry, first degree of freedom of the F distribution (numerator). Any value in the array NDF1 must be greater or equal to 1.

The size of NDF1 must be size(NDF1) = size(F) .

NDF2 (INPUT) integer(i4b), dimension(:) On entry, second degree of freedom of the F distribution (denominator). Any value in the array NDF2 must be greater or equal to 1.

The size of NDF2 must be size(NDF2) = size(F) .

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of F is calculated.
- UPPER = false : probability to the left of F is calculated.

Further Details

This function accepts only integer values of degree of freedom and uses a normal approximation. This normal approximation is not accurate for small values of degrees of freedom.

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Peizer, D.B., and Pratt, J.W., 1968:** A normal approximation for Binomial, F, Beta, and . . . , (formula 2.24a). J.A.S.A., Vol. 63, 1457-1483

6.15.84 `function probf ( f, ndf1, ndf2, upper )`

Purpose

Evaluates the F distribution function with degrees of freedom NDF1 and NDF2 from F(i,j) to infinity if UPPER is true or from zero to F(i,j) if UPPER is false, for i=1 to size(F,1) and j=1 to size(F,2). In other words, if:

- UPPER = true : PROBF(i,j) = prob(U > F(i,j)) ,
- UPPER = false : PROBF(i,j) = prob(U < F(i,j)) ,

, for U = Fisher(NDF1,NDF2) and i=1 to size(F,1) and j=1 to size(F,2).

Arguments

F (INPUT) real(stnd), dimension(:,:) On entry, upper or lower limits of integration. F(i,j) must be greater or equal to zero for i=1 to size(F,1) and j=1 to size(F,2).

NDF1 (INPUT) integer(i4b) On entry, first degree of freedom of the F distribution (numerator). NDF1 must be greater or equal to 1.

NDF2 (INPUT) integer(i4b) On entry, second degree of freedom of the F distribution (denominator). NDF2 must be greater or equal to 1.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of F is calculated.
- UPPER = false : probability to the left of F is calculated.

Further Details

This function accepts only integer values of degree of freedom and uses a normal approximation. This normal approximation is not accurate for small values of degrees of freedom.

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Peizer, D.B., and Pratt, J.W., 1968:** A normal approximation for Binomial, F, Beta, and . . . , (formula 2.24a). J.A.S.A., Vol. 63, 1457-1483

6.15.85 function probf (f, ndf1, ndf2, upper)

Purpose

Evaluates the F distribution function with degrees of freedom NDF1 and NDF2 from F(i,j) to infinity if UPPER is true or from zero to F(i,j) if UPPER is false, for i=1 to size(F,1) and j=1 to size(F,2). In other words, if:

- UPPER = true : PROBF(i,j) = prob(U > F(i,j)) ,
- UPPER = false : PROBF(i,j) = prob(U < F(i,j)) ,

, for U = Fisher(NDF1(i,j),NDF2(i,j)) and i=1 to size(F,1) and j=1 to size(F,2).

Arguments

F (INPUT) real(stnd), dimension(:,:) On entry, upper or lower limits of integration. F(i,j) must be greater or equal to zero for i=1 to size(F,1) and j=1 to size(F,2).

NDF1 (INPUT) integer(i4b), dimension(:,:) On entry, first degree of freedom of the F distribution (numerator). Any value in the array NDF1 must be greater or equal to 1.

The shape of NDF1 must verify:

- size(NDF1,1) = size(F,1)
- size(NDF1,2) = size(F,2) .

NDF2 (INPUT) integer(i4b), dimension(:,:) On entry, second degree of freedom of the F distribution (denominator). Any value in the array NDF2 must be greater or equal to 1.

The shape of NDF2 must verify:

- size(NDF2,1) = size(F,1)
- size(NDF2,2) = size(F,2) .

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of F is calculated.
- UPPER = false : probability to the left of F is calculated.

Further Details

This function accepts only integer values of degree of freedom and uses a normal approximation. This normal approximation is not accurate for small values of degrees of freedom.

This function is parallelized if OPENMP is used.

This function is adapted from:

(1) **Peizer, D.B., and Pratt, J.W., 1968:** A normal approximation for Binomial, F, Beta, and . . . , (formula 2.24a). J.A.S.A., Vol. 63, 1457-1483

6.15.86 function probf2 (f, df1, df2, upper, beta, acu, maxiter, failure)

Purpose

Evaluates the F distribution function with degrees of freedom DF1 and DF2 (integer or fractional degrees of freedom) from F to infinity if UPPER is true or from zero to F if UPPER is false. In other words, if:

- UPPER = true : PROBF2 = prob(U > F) ,
- UPPER = false : PROBF2 = prob(U <= F) ,

, for U = Fisher(DF1,DF2).

For given arguments F (0<=F), DF1 (DF1>0), DF2 (DF2>0), PROBF2 returns the probability that a random variable from an F distribution having DF1 and DF2 degrees of freedom will be less than or equal to F (if UPPER is false) or greater than F (if UPPER is true).

Arguments

F (INPUT) real(stnd) On entry, upper or lower limit of integration. F must be greater or equal to zero.

DF1 (INPUT) real(stnd) On entry, first degree of freedom of the F distribution (numerator). DF1 must be greater than zero.

DF2 (INPUT) real(stnd) On entry, second degree of freedom of the F distribution (denominator). DF2 must be greater than zero.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of F is calculated.
- UPPER = false : probability to the left of F is calculated.

BETA (INPUT, OPTIONAL) real(stnd) On entry, the logarithm of the complete beta function BETA(0.5 * DF1,0.5 * DF2).

If BETA is not given, the logarithm of the beta function is computed with the help of function LNGAMMA.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result, when computing the incomplete beta function. The "integrating" process for evaluating the incomplete beta function is terminated when the relative contribution to the integral is not greater than the value of ACU. ACU is a small strictly positive integer.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the power series representation of the incomplete Beta integral.

The default value is 2000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process of the incomplete Beta integral did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

Further Details

This function invoked the BETA distribution function (e.g. function PROBBETA) for computing the probability associated with F and is much more accurate than PROBF function.

6.15.87 function probf2 (f, df1, df2, upper, beta, acu, maxiter, failure)

Purpose

Evaluates the F distribution function with degrees of freedom DF1 and DF2 (integer or fractional degrees of freedom) from F(i) to infinity if UPPER is true or from zero to F(i) if UPPER is false, for i=1 to size(F). In other words, if:

- UPPER = true : PROBF(i) = prob(U > F(i)) ,
- UPPER = false : PROBF(i) = prob(U < F(i)) ,

, for U = Fisher(DF1,DF2) and i=1 to size(F).

For given arguments F(:) (0<=F(:)), DF1 (DF1>0), DF2 (DF2>0), PROBF2 returns the probability that a random variable (vector) from an F distribution having DF1 and DF2 degrees of freedom will be less than or equal to F (if UPPER is false) or greater than F (if UPPER is true).

Arguments

F (INPUT) real(stnd), dimension(:) On entry, upper or lower limit of integration. F(i) must be greater or equal to zero for i=1 to size(F).

DF1 (INPUT) real(stnd) On entry, first degree of freedom of the F distribution (numerator). DF1 must be greater than zero.

DF2 (INPUT) real(stnd) On entry, second degree of freedom of the F distribution (denominator). DF2 must be greater than zero.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of F is calculated.
- UPPER = false : probability to the left of F is calculated.

BETA (INPUT, OPTIONAL) real(stnd) On entry, the logarithm of the complete beta function BETA(0.5 * DF1,0.5 * DF2).

If BETA is not given, the logarithm of the beta function is computed with the help of function LNGAMMA.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result, when computing the incomplete beta function. The "integrating" process for evaluating the incomplete beta function is terminated when the relative contribution to the integral is not greater than the value of ACU. ACU is a small strictly positive integer.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the power series representation of the incomplete Beta integral.

The default value is 2000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process of the incomplete Beta integral did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

Further Details

This function invoked the BETA distribution function (e.g. function PROBBETA) for computing the probability associated with F and is much more accurate than PROBF function.

6.15.88 function probf2 (f, df1, df2, upper, acu, maxiter, failure)

Purpose

Evaluates the F distribution function with degrees of freedom DF1(i) and DF2(i) (integer or fractional degrees of freedom) from F(i) to infinity if UPPER is true or from zero to F(i) if UPPER is false, for i=1 to size(F). In other words, if:

- UPPER = true : PROBF(i) = prob(U > F(i)) ,
- UPPER = false : PROBF(i) = prob(U < F(i)) ,

, for U = Fisher(DF1(i),DF2(i)) and i=1 to size(F).

For given arguments F(:) (0<=F(:)), DF1(:) (DF1(:)>0), DF2(:) (DF2(:)>0), PROBF2 returns the probability that a random variable (vector) from an F distribution having DF1(:) and DF2(:) degrees of freedom will be less than or equal to F(:) (if UPPER is false) or greater than F(:) (if UPPER is true).

Arguments

F (INPUT) real(stnd), dimension(:) On entry, upper or lower limit of integration. F(i) must be greater or equal to zero for i=1 to size(F).

DF1 (INPUT) real(stnd), dimension(:) On entry, first degree of freedom of the F distribution (numerator). Any value in the array DF1(:) must be greater than zero.

The size of DF1 must verify size(DF1) = size(F) .

DF2 (INPUT) real(stnd), dimension(:) On entry, second degree of freedom of the F distribution (denominator). Any value in the array DF2(:) must be greater than zero.

The size of DF2 must verify size(DF2) = size(F) .

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of F is calculated.
- UPPER = false : probability to the left of F is calculated.

BETA (INPUT, OPTIONAL) real(stnd) On entry, the logarithm of the complete beta function BETA(0.5 * DF1,0.5 * DF2).

If BETA is not given, the logarithm of the beta function is computed with the help of function LNGAMMA.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result, when computing the incomplete beta function. The "integrating" process for evaluating the incomplete beta function is terminated when the relative contribution to the integral is not greater than the value of ACU. ACU is a small strictly positive integer.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the power series representation of the incomplete Beta integral.

The default value is 2000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process of the incomplete Beta integral did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

Further Details

This function invoked the BETA distribution function (e.g. function PROBBETA) for computing the probability associated with F and is much more accurate than PROBF function.

The function is parallelized if OPENMP is used.

6.15.89 function probf2 (f, df1, df2, upper, beta, acu, maxiter, failure)

Purpose

Evaluates the F distribution function with degrees of freedom DF1 and DF2 (integer or fractional degrees of freedom) from F(i,j) to infinity if UPPER is true or from zero to F(i,j) if UPPER is false, for i=1 to size(F,1) and j=1 to size(F,2):

if UPPER = true : PROBF(i, j) = prob(U > F(i,j)) , if UPPER = false : PROBF(i, j) = prob(U < F(i,j)) ,

, for U = Fisher(DF1,DF2) and i=1 to size(F,1) and j=1 to size(F,2).

For given arguments F(:,:) (0<=F(:,:)), DF1 (DF1>0), DF2 (DF2>0), PROBF2 returns the probability that a random variable (matrix) from an F distribution having DF1 and DF2 degrees of freedom will be less than or equal to F (if UPPER is false) or greater than F (if UPPER is true).

Arguments

F (INPUT) real(stnd), dimension(:,:) On entry, upper or lower limit of integration. F(i,j) must be greater or equal to zero for i=1 to size(F,1) and j=1 to size(F,2).

DF1 (INPUT) real(stnd) On entry, first degree of freedom of the F distribution (numerator). DF1 must be greater than zero.

DF2 (INPUT) real(stnd) On entry, second degree of freedom of the F distribution (denominator). DF2 must be greater than zero.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of F is calculated.
- UPPER = false : probability to the left of F is calculated.

BETA (INPUT, OPTIONAL) real(stnd) On entry, the logarithm of the complete beta function BETA(0.5 * DF1,0.5 * DF2).

If BETA is not given, the logarithm of the beta function is computed with the help of function LNGAMMA.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result, when computing the incomplete beta function. The "integrating" process for evaluating the incomplete beta function is terminated when the relative contribution to the integral is not greater than the value of ACU. ACU is a small strictly positive integer.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the power series representation of the incomplete Beta integral.

The default value is 2000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process of the incomplete Beta integral did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

Further Details

This function invoked the BETA distribution function (e.g. function PROBBETA) for computing the probability associated with F and is much more accurate than PROBF function.

6.15.90 function probf2 (f, df1, df2, upper, acu, maxiter, failure)

Purpose

Evaluates the F distribution function with degrees of freedom DF1(i,j) and DF2(i,j) (integer or fractional degrees of freedom) from F(i,j) to infinity if UPPER is true or from zero to F(i,j) if UPPER is false, for i=1 to size(F,1) and j=1 to size(F,2). In other words, if:

- UPPER = true : PROBF(i, j) = prob(U > F(i,j)) ,
- UPPER = false : PROBF(i, j) = prob(U < F(i,j)) ,

, for U = Fisher(DF1(i,j),DF2(i,j)) and i=1 to size(F,1) and j=1 to size(F,2).

For given arguments F(:,:) (0<=F(:,:)), DF1(:,:) (DF1(:,:)>0), DF2(:,:) (DF2(:,:)>0), PROBF2 returns the probability that a random variable (matrix) from an F distribution having DF1(:,:) and DF2(:,:) degrees of freedom will be less than or equal to F(:,:) (if UPPER is false) or greater than F (if UPPER is true).

Arguments

F (INPUT) real(stnd), dimension(:,:) On entry, upper or lower limit of integration. F(i,j) must be greater or equal to zero for i=1 to size(F,1) and j=1 to size(F,2).

DF1 (INPUT) real(stnd), dimension(:,:) On entry, first degree of freedom of the F distribution (numerator). Any value in the array DF1(:,:) must be greater than zero.

The shape of DF1 must verify:

- size(DF1,1) = size(F,1)
- size(DF1,2) = size(F,2) .

DF2 (INPUT) real(stnd), dimension(:,:) On entry, second degree of freedom of the F distribution (denominator). Any value in the array DF2(:,:) must be greater than zero.

The shape of DF2 must verify:

- size(DF2,1) = size(F,1)
- size(DF2,2) = size(F,2) .

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : probability to the right of F is calculated.
- UPPER = false : probability to the left of F is calculated.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result, when computing the incomplete beta function. The "integrating" process for evaluating the incomplete beta function is terminated when the relative contribution to the integral is not greater than the value of ACU. ACU is a small strictly positive integer.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the power series representation of the incomplete Beta integral.

The default value is 2000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process of the incomplete Beta integral did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

Further Details

This function invoked the BETA distribution function (e.g. function PROBBETA) for computing the probability associated with F and is much more accurate than PROBF function.

The function is parallelized if OPENMP is used.

6.15.91 `function pinvf2 ( p, df1, df2, beta, acu, maxiter )`

Purpose

Evaluates the inverse F probability distribution function with degrees of freedom DF1 and DF2 (integer or fractional degrees of freedom).

For given arguments P (0<=P<=1), DF1 (DF1>0.2), DF2 (DF2>0.2), PINVF2 returns the value F such that the probability that a random variable distributed as F(DF1,DF2) is less than or equal to F is P.

Arguments

P (INPUT) real(stnd) On entry, input probability in the inclusive range (0,1).

DF1 (INPUT) real(stnd) On entry, first degree of freedom of the F distribution (numerator). DF1 must be greater than 0.2.

DF2 (INPUT) real(stnd) On entry, second degree of freedom of the F distribution (denominator). DF2 must be greater than 0.2.

BETA (INPUT, OPTIONAL) real(stnd) On entry, the logarithm of the complete beta function BETA(0.5 * DF1,0.5 * DF2).

If BETA is not given, the logarithm of the beta function is computed with the help of function LNGAMMA.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result, when computing the incomplete beta function. The "integrating" process for evaluating the incomplete beta function is terminated when the relative contribution to the integral is not greater than the value of ACU. ACU is a small strictly positive integer.

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the power series representation of the incomplete Beta integral.

See the description of the PROBBETA function for more details on this argument.

The default value is 2000.

Further Details

This function invoked the inverse BETA distribution function (e.g. function PINVBETA) for computing the value F associated with the probability P.

This function is not very accurate for small values of DF1 and/or DF2 (e.g. less than 1).

6.15.92 function probbinom (prob, n, k, upper, beta, acu, maxiter, failure)

Purpose

Evaluates the cumulative binomial probability distribution function for a positive real argument PROB between 0 and 1, a strictly positive integer N and a positive integer K less than or equal to N.

PROBBINOM computes the probability that an event occurring with probability PROB per trial, will occur K or more times in N independent trials if UPPER is true, or will occur K or less times in N independent trials if UPPER is false.

This probability is estimated with the help of the incomplete beta fonction, as computed by PROBBETA function, and the optional arguments BETA, ACU, MAXITER and FAILURE are passed directly to the PROBBETA function if these arguments are present.

Arguments

PROB (INPUT) real(stnd) On entry, a positive real argument PROB which is the probability of success on each trial for the given binomial probability distribution. PROB must be greater or equal to zero and less than or equal to 1.

N (INPUT) integer(i4b) On entry, a strictly positive integer argument which is the total number of Bernoulli trials for the given binomial probability distribution. N must be greater than zero.

K (INPUT) integer(i4b) On entry, a positive integer argument which is the minimal (if UPPER is true) or maximum (if UPPER is false) number of success for the N Bernoulli trials, for which we want to compute the cumulative probability following the binomial probability distribution specified by PROB. K must be greater or equal to zero and less or equal to N.

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : the probability that the number of success is greater than or equal to K in N trials is computed.

- UPPER = false : the probability that the number of success is less than or equal to K in N trials is computed.

BETA (INPUT, OPTIONAL) real(stnd) On entry, the logarithm of the complete beta function BETA(K,N-K+1) if UPPER is true, or BETA(N-K,K+1) if UPPER is false.

If BETA is not given, the logarithm of the beta function is computed with the help of function LNGAMMA.

ACU (INPUT, OPTIONAL) real(stnd) On entry, the desired accuracy of the result when evaluating the incomplete beta fonction. The "integrating" process is terminated when both the absolute and relative contributions to the integral is not greater than the value of ACU.

ACU is a small strictly positive integer. If the number of decimal digits' accuracy required is r, ACU should be set to 10**-(r+1).

The default value for ACU is epsilon(ACU).

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, MAXITER controls the maximum number of iterations when evaluating the power series representation of the incomplete Beta integral.

The default value is 2000.

FAILURE (INPUT, OPTIONAL) logical(lgl) On entry, if FAILURE is set to true, the values for which the "integrating" process did not converge to the desired accuracy are set to -1. The algorithm fails to converge if the number of iterations exceeds MAXITER.

The default value is false.

Further Details

The cumulative binomial probability is computed with the help of the incomplete Beta function PROBBETA as follow:

If UPPER is true, the probability of an event will occur K or more times in N independant trials, if its probability per trial is PROB, is computed as PROBBETA(PROB, K, N-K+1).

If UPPER is false, the probability of an event will occur K or less times in N independant trials, if its probability per trial is PROB, is computed as PROBBETA(1-PROB, N-K, K+1).

6.15.93 `function rangen ( x, n )`

Purpose

Evaluates the probability of the normal range given X, the upper limit of integration, and N, the sample size.

In other words, this function evaluates the probability that the standardized difference between the maximum and the minimum on a sample will be less than X for a normal (e.g. gaussian) sample of size N.

Arguments

X (INPUT) real(stnd) On entry, upper limit of integration. X must be greater than zero.

N (INPUT) integer(i4b) On entry, the sample size. N must be greater than 1.

Further Details

This function is adapted from:

(1) **Barnard, J., 1978:** Algorithm AS126: Probability Integral of the normal range, Applied Statistics, Vol. 27, No. 2, pp.197-198

6.15.94 `function rangen ( x, n )`

Purpose

Evaluates probabilities of the normal range given X(:), the upper limits of integration, and N, the sample size.

In other words, this function evaluates the probabilities that the standardized difference between the maximum and the minimum on a sample will be less than X(:) for a normal (e.g. gaussian) sample of size N.

Arguments

X (INPUT) real(stnd), dimension(:) On entry, upper limits of integration. All elements in X(:) must be greater than zero.

N (INPUT) integer(i4b) On entry, the sample size. N must be greater than 1.

Further Details

This function is adapted from:

(1) **Barnard, J., 1978:** Algorithm AS126: Probability Integral of the normal range, Applied Statistics, Vol. 27, No. 2, pp.197-198

6.16 Module_QR_Procedures

Copyright 2025 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING SUBROUTINES AND FUNCTIONS FOR COMPUTING QR, LQ DECOMPOSITIONS AND RELATED DECOMPOSITIONS.

LATEST REVISION : 03/09/2025

6.16.1 subroutine lq_cmp (mat, diagl, tau, use_qr)

Purpose

LQ_CMP computes a LQ factorization of a real m-by-n matrix MAT :

MAT = L * Q

where Q is orthogonal and L is lower trapezoidal (lower triangular if m<=n).

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real matrix to be decomposed.

On exit, the elements below the diagonal of the array contain the corresponding elements of L. The elements on and above the diagonal, with the array TAU, represent the orthogonal matrix Q as a product of elementary reflectors, see Further Details.

DIAGL (OUTPUT) real(stnd), dimension(:) On exit, the diagonal elements of the matrix L.

The size of DIAGL must be min(size(MAT,1) , size(MAT,2)).

TAU (OUTPUT) real(stnd), dimension(:) On exit, the scalars factors of the elementary reflectors. see Further Details.

The size of TAU must be min(size(MAT,1) , size(MAT,2)).

USE_QR (INPUT, OPTIONAL) logical(lgl) If the optional argument USE_QR is set to true, the input matrix is transposed, a fast QR decomposition is used for computing the LQ decomposition and the results are transposed again in the input matrix.

The default is true.

Further Details

The matrix Q is represented as a product of elementary reflectors

Q = H(k) * . . . * H(2) * H(1), where k = min(size(MAT,1) , size(MAT,2))

Each H(i) has the form

H(i) = I + tau * (v * v') ,

where tau is a real scalar and v is a real n-element vector with v(1:i-1) = 0. v(i:n) is stored on exit in MAT(i,i:n) and tau in TAU(i).

A blocked algorithm is used for computing the LQ factorization. Furthermore, the computations are parallelized if OPENMP is used.

On exit of LQ_CMP, the orthogonal matrix Q (or its first rows) can be computed explicitly by a call to subroutine ORTHO_GEN_LQ with arguments MAT and TAU.

For further details on the LQ factorization and its use or the blocked algorithm used here, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(4) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.16.2 `subroutine ortho_gen_lq ( mat, tau, use_qr )`

Purpose

ORTHO_GEN_LQ generates an m-by-n real matrix with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

Q = H(k) * . . . * H(2) * H(1)

as returned by LQ_CMP.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,. . . ,k, as returned by LQ_CMP in the first k rows of its array argument MAT.

On exit, the first m rows of Q.

The shape of MAT must verify: size(MAT, 1) <= size(MAT, 2).

TAU (INPUT) real(stnd), dimension(:) TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by LQ_CMP. The size of TAU determines the number k of elementary reflectors whose product defines the matrix Q.

The size of TAU must verify: size(TAU) <= size(MAT, 1) .

USE_QR (INPUT, OPTIONAL) logical(lgl) If the optional argument USE_QR is set to true, the input matrix is transposed and a fast QR type algorithm is used for computing the first m rows of Q and the results are transposed again in the input matrix.

The default is true.

Further Details

This subroutine used a blocked algorithm for agregating the Householder transformations (e.g. the elementary reflectors) stored in the upper triangle of MAT and generating the orthogonal matrix Q of the LQ factorization.

Furthermore, the computations are parallelized if OPENMP is used.

For further details on the LQ factorization and its use or the blocked algorithm, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(4) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.16.3 subroutine apply_q_lq (mat, tau, c, left, trans, use_qr)

Purpose

APPLY_Q_LQ overwrites the general real m-by-n matrix C with

Q * C if LEFT = true and TRANS = false, or

Q' * C if LEFT = true and TRANS = true, or

C * Q if LEFT = false and TRANS = false, or

C * Q' if LEFT = false and TRANS = true,

where Q is a real orthogonal matrix defined as the product of k elementary reflectors

Q = H(k) * ... * H(2) * H(1)

as returned by LQ_CMP. Q is of order m if LEFT = true and of order n if LEFT = false.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by LQ_CMP in the first k rows of its array argument MAT. MAT is not modified by the routine.

TAU (INPUT) real(stnd), dimension(:) TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by LQ_CMP. The size of TAU determines the number k of elementary reflectors whose product defines the matrix Q.

The size of TAU must verify: size(TAU) <= min(size(MAT,1) , size(MAT,2)) .

C (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m by n matrix C.

On exit, C is overwritten by Q * C or Q' * C or C * Q' or C * Q.

The shape of C must verify:

- if LEFT = true, size(C, 1) = size(MAT, 2)
- if LEFT = false, size(C, 2) = size(MAT, 2) .

LEFT (INPUT) logical(lgl) If:

- LEFT = true : apply Q or Q' from the left
- LEFT = false : apply Q or Q' from the right

TRANS (INPUT) logical(lgl) If:

- TRANS = false : apply Q (no transpose)
- TRANS = true : apply Q' (transpose)

USE_QR (INPUT, OPTIONAL) logical(lgl) If the optional argument USE_QR is set to true, the input matrix MAT is transposed and a fast QR type algorithm is used for applying Q or its transpose to the matrix C.

The default is true.

Further Details

This subroutine is adapted from the routine DORML2 in LAPACK.

This subroutine used a blocked algorithm for agregating the Householder transformations (e.g. the elementary reflectors) stored in the upper triangle of MAT and applying the orthogonal matrix Q of the LQ factorization to the real m-by-n matrix C.

Furthermore, the computations are parallelized if OPENMP is used.

For further details on the LQ factorization and its use or the blocked algorithm used here, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(4) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.16.4 `subroutine apply_q_lq ( mat, tau, c, trans )`

Purpose

APPLY_Q_LQ overwrites the general real m vector C with

Q * C if TRANS = false, or

Q' * C if TRANS = true,

where Q is a real orthogonal matrix defined as the product of k elementary reflectors

Q = H(k) * ... * H(2) * H(1)

as returned by LQ_CMP. Q is of order m.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the i-th row must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by LQ_CMP in the first k rows of its array argument MAT. MAT is not modified by the routine.

TAU (INPUT) real(stnd), dimension(:) TAU(i) must contain the scalar factor of the elementary reflector H(i), as returned by LQ_CMP. The size of TAU determines the number k of elementary reflectors whose product defines the matrix Q.

The size of TAU must verify: size(TAU) <= min(size(MAT,1) , size(MAT,2)) .

C (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the m vector C.

On exit, C is overwritten by Q * C or Q' * C.

The shape of C must verify: size(C) = size(MAT, 2).

TRANS (INPUT) logical(lgl) If:

- TRANS = false : apply Q (no transpose)

- TRANS = true : apply Q' (transpose)

Further Details

This subroutine is adapted from the routine DORML2 in LAPACK.

For further details on the LQ factorization and its use or the algorithm used here, see

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.16.5 `subroutine qr_cmp ( mat, diagr, beta )`

Purpose

QR_CMP computes a QR factorization of a real m-by-n matrix MAT :

MAT = Q * R

where Q is orthogonal and R is upper trapezoidal (upper triangular if m>=n).

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real matrix to be decomposed.

On exit, the elements above the diagonal of the array contain the corresponding elements of R. The elements on and below the diagonal, with the array BETA, represent the orthogonal matrix Q as a product of elementary reflectors, see Further Details.

DIAGR (OUTPUT) real(stnd), dimension(:) On exit, the diagonal elements of the matrix R.

The size of DIAGR must verify: size(DIAGR) <= min(size(MAT,1) , size(MAT,2))

BETA (OUTPUT) real(stnd), dimension(:) On exit, the scalars factors of the elementary reflectors.

The size of BETA must verify:

- size(BETA) = size(DIAGR) <= min(size(MAT,1) , size(MAT,2))

See Further Details.

Further Details

The matrix Q is represented as a product of elementary reflectors

Q = H(1) * H(2) * . . . * H(k), where k = size(BETA) = size(DIAGR)

Each H(i) has the form

H(i) = I + beta * (v * v') ,

where beta is a real scalar and v is a real m-element vector with v(1:i-1) = 0. v(i:m) is stored on exit in MAT(i:m,i) and beta in BETA(i).

A blocked algorithm is used for computing the QR factorization. Furthermore, the computations are parallelized if OPENMP is used.

On exit of QR_CMP, the orthogonal matrix Q (or its first columns) can be computed explicitly by a call to subroutines ORTHO_GEN_QR or ORTHO_GEN_QR_TS with arguments MAT and BETA.

For further details on the QR factorization and its use or the blocked algorithm used here, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(4) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.16.6 `subroutine qr_cmp_ts ( mat, diagr, beta )`

Purpose

QR_CMP_TS computes a QR factorization of a real m-by-n matrix MAT :

MAT = Q * R

where Q is orthogonal and R is upper trapezoidal (upper triangular if m>=n).

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real matrix to be decomposed.

On exit, the elements above the diagonal of the array contain the corresponding elements of R. The elements on and below the diagonal, with the array BETA, represent the orthogonal matrix Q as a product of elementary reflectors, see Further Details.

DIAGR (OUTPUT) real(stnd), dimension(:) On exit, the diagonal elements of the matrix R.

The size of DIAGR must verify: size(DIAGR) <= min(size(MAT,1) , size(MAT,2))

BETA (OUTPUT) real(stnd), dimension(:) On exit, the scalars factors of the elementary reflectors.

The size of BETA must verify:

- size(BETA) = size(DIAGR) <= min(size(MAT,1) , size(MAT,2))

See Further Details.

Further Details

The matrix Q is represented as a product of elementary reflectors

Q = H(1) * H(2) * . . . * H(k), where k = size(BETA) = size(DIAGR)

Each H(i) has the form

H(i) = I + beta * (v * v') ,

where beta is a real scalar and v is a real m-element vector with v(1:i-1) = 0. v(i:m) is stored on exit in MAT(i:m,i) and beta in BETA(i).

A blocked algorithm is used for computing the QR factorization. Furthermore, the computations are parallelized if OPENMP is used.

On exit of QR_CMP_TS, the orthogonal matrix Q (or its first columns) can be computed explicitly by a call to subroutine ORTHO_GEN_QR or ORTHO_GEN_QR_TS with arguments MAT and BETA.

This subroutine performs exactly the same task as subroutine QR_CMP, but will be sometime more robust (and efficient) when OpenMP is used for huge tall and skinny m-by-n matrices MAT, as QR_CMP_TS uses only BLAS1 kernels instead of BLAS3 and BLAS2 kernels as in QR_CMP in the blocked algorithm.

For further details on the QR factorization and its use or the blocked algorithm used here, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(4) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.16.7 subroutine qrfac (name_proc, syst, kfix, krank, min_norm, diagr, beta, h, tol, ip)

Purpose

QRFAC is a low-level subroutine for computing a QR or complete orthogonal factorization of the array section SYST(:m,:n) where m=size(SYST,1) and n<=size(SYST,2).

The routine first computes a QR factorization with column pivoting of SYST(:m,:n):

SYST(:m,:n) * P = Q * R

, where P is an n-by-n permutation matrix, R is an upper triangular or trapezoidal (i.e., if n>m) matrix and Q is a m-by-m orthogonal matrix.

The orthogonal transformation Q is then applied to the array section SYST(:m,n+1:):

SYST(:m,n+1:) = Q * B

Then, the rank of SYST(:m,:n) is determined by finding the submatrix R11 of R which is defined as the largest leading submatrix whose estimated condition number, in the 1-norm, is less than 1/TOL or such that abs(R11[j,j])>0 if TOL is absent. The order of R11, krank, is an estimate of the rank of SYST(:m,:n).

This leads implicitly to the following partition of Q:

[Q1 Q2]

where Q1 is an m-by-krank orthonormal matrix and Q2 is m-by-(m-krank) orthonormal matrix orthogonal to Q1, and to the following corresponding partition of R:

[R11 R12]

[R21 R22]

where R11 is a krank-by-krank triangular matrix, R21 is zero by construction, R12 is a full krank-by-(n-krank) matrix and R22 is a full (m-krank)-by-(n-krank) matrix.

Q1 and Q2 are, respectively, orthonormal bases of the range and null space of SYST(:m,:n).

If MIN_NORM=true, R22 is considered to be negligible and R12 is annihilated by orthogonal transformations from the right, arriving at the complete orthogonal factorization:

SYST(:m,:n) * P = Q * T * Z

, where P is a n-by-n permutation matrix, Q is a m-by-m orthogonal matrix, Z is a n-by-n orthogonal matrix and T is a m-by-n matrix, which has the form:

[T11 T12]

[T21 T22]

Here T21 (=R21) and T12 are all zero, T22 (=R22) is considered to be negligible and T11 is a krank-by-krank upper triangular matrix.

On exit, P is stored compactly in the vector argument IP, krank is stored in the scalar argument KRANK and if:

- MIN_NORM=false, QRFAC computes Q and submatrices R11 and R12. Submatrices R11, and R12 are stored in the array section SYST(:krank,:n) and the array argument DIAGR. Q is stored compactly in factored form in the array section SYST(:m,:n) (in its lower triangle) and the array argument BETA.
- MIN_NORM=true, QRFAC computes Q, Z and submatrice T11. Submatrice T11 is stored in the array section SYST(:krank,:krank) and the array argument DIAGR. Q is stored compactly in factored form in the array section SYST(:m,:n) (in its lower triangle) and the array argument BETA. Z is stored compactly in factored form in the array sections SYST(:krank,krank+1:n) and H(:krank).

See Further Details for more information.

Arguments

NAME_PROC (INPUT) character(len=*) Name of the subroutine calling QRFAC.

SYST (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real m-by-n matrix to be decomposed and, eventually, the the right hand side matrix of an associated least squares problem in the array section SYST(:m,n+1:).

On exit, SYST(:m,1:n) has been overwritten by details of its QR or (complete) orthogonal factorization and B is stored in SYST(:m,n+1:).

See Further Details.

KFIX (INPUT) integer(i4b) On entry:

- KFIX=k, implies that the first k columns of SYST(:m,:n) are to be forced into the basis. Pivoting is performed only on the last n-k columns of SYST(:m,:n) if k<min(m,n).
- KFIX<=0 can be used when pivoting is desired on all columns of SYST(:m,:n).
- If KFIX<min(m,n) then the optional argument IP must be present to store the permutation matrix P.
- When KFIX>=min(m,n) is used, pivoting is not performed. This is appropriate when SYST(:m,:n) is known to be of full rank.

KRANK (INPUT/OUTPUT) integer(i4b) On entry, KRANK=n . KRANK must verify: KRANK <= size(SYST, 2).

On exit, KRANK contains the effective rank of SYST(:m,:n), i.e., the order of the submatrix R11, krank. This is the same as the order of the submatrix T11 in the complete orthogonal factorization of SYST(:m,:n).

MIN_NORM (INPUT) logical(lgl) On entry:

- MIN_NORM=true indicates that a complete orthogonal factorization of SYST(:m,:n) must be computed.
- MIN_NORM=false indicates that only a QR factorization (eventually with column pivoting if KFIX<min(m,n)) of SYST(:m,:n) must be computed.

DIAGR (OUTPUT) real(stnd), dimension(:) On exit, the diagonal elements of the matrix R if MIN_NORM=false or the diagonal elements of the matrix T11 if MIN_NORM=true.

The diagonal elements of R are stored in DIAGR(:min(m,n)) and the diagonal elements of T11 are stored in DIAGR(:krank) and krank is stored in the real argument KRANK on exit.

See Further Details.

The size of DIAGR must verify:

- size(DIAGR) >= min(m , n) = min(size(SYST,1) , KRANK).

BETA (OUTPUT) real(stnd), dimension(:) On exit, the scalars factors of the elementary reflectors defining Q.

See Further Details.

The size of BETA must verify:

- size(BETA) >= min(m , n) = min(size(SYST,1) , KRANK).

H (OUTPUT) real(stnd), dimension(:) On exit, if MIN_NORM=true, the scalars factors of the elementary reflectors defining Z in the complete orthogonal factorization of MAT are stored in H(:krank). On exit, krank is output in the real argument KRANK.

See Further Details.

The size of H must verify: size(H) >= n = KRANK .

TOL (INPUT/OUTPUT, OPTIONAL) real(stnd) If TOL is present and is in [0,1[, then:

- On entry, the calculations to determine the condition number of of SYST(:m,:n) in the 1-norm are performed. Then, TOL is used to determine the effective rank of SYST(:m,:n), which is defined as the order of the largest leading triangular submatrix R11 in the QR factorization with pivoting of SYST(:m,:n), whose estimated condition number is less than 1/TOL. If TOL=0 is specified the numerical rank of SYST(:m,:n) is determined.
- On exit, the reciprocal of the condition number is returned in TOL.

If TOL is not specified or is outside [0,1[:

- The calculations to determine the condition number of MAT are not performed and crude tests on R(j,j) are done to determine the rank of MAT. If TOL is present, it is not changed.

IP (OUTPUT, OPTIONAL) integer(i4b), dimension(:) On exit, if IP(j)=k, then the j-th column of SYST(:m,:n) * P was the k-th column of SYST(:m,:n).

IP must be present if KFIX < min(m , n) = min(SYST,1), KRANK).

The size of IP must verify: size(IP) >= n = KRANK.

Further Details

QRFAC is called by the subroutines QR_CMP2, LLSQ_QR_SOLVE, LLSQ_QR_SOLVE2 and SOLVE_LLSQ in modules QR_Procedures and LLSQ_Procedures.

Since QRFAC is a low level subroutine, no checking of the correctness of the dimensions of the array arguments is performed inside of the subroutine and such checking must be done before calling QRFAC.

The matrix Q is represented as a product of elementary reflectors

Q = H(1) * H(2) * . . . * H(k), where k = min(m , n).

Each H(i) has the form

H(i) = I + beta * (v * v') ,

where beta is a real scalar and v is a real m-element vector with v(1:i-1) = 0. v(i:m) is stored on exit in SYST(i:m,i) and beta in BETA(i).

On exit of QRFAC, the orthonormal matrix Q (or its first n columns) can be computed explicitly by a call to subroutine ORTHO_GEN_QR with arguments MAT and BETA.

The matrix P is represented in the array IP (if present) as follows: If IP(j) = i then the jth column of P is the ith canonical unit vector.

If MIN_NORM=false, a QR factorization with column pivoting of SYST(:m,:n) is computed and, on exit:

- The elements above the diagonal of the array SYST(:krank,:krank) contain the corresponding elements of the triangular matrix R11.
- The elements of the diagonal of R11 are stored in the array DIAGR.
- The submatrix R12 is stored in SYST(:krank,krank+1:n).
- krank is stored in the real argument KRANK.

If MIN_NORM=true, a complete orthogonal factorization of SYST(:m,:n) is computed. The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), which is used to introduce zeros into the kth row of R, is given in the form

[I 0]

[0 T(k)]

where

T(k) = I + tau * (u(k) * u(k)') and u(k)' = (1 0 z(k))

tau is a scalar, u(k) is a n-k+1 vector and z(k) is an (n-krank) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of R12.

The Z n-by-n orthogonal matrix is given by

Z = Z(1) * Z(2) * . . . * Z(krank)

On exit, if MIN_NORM=true:

- The scalar tau defining T(k) is returned in the kth element of H and the vector u(k) in the kth row of SYST, such that the elements of z(k) are in SYST(k,krank+1:n). The other elements of u(k) are not stored.
- The elements above the diagonal of the array section SYST(:krank,:krank) contain the corresponding elements of the triangular matrix T11. The elements of the diagonal of T11 are stored in the array DIAGR.
- krank is stored in the real argument KRANK.

The computations are parallelized if OPENMP is used. QRFAC uses a blocked "BLAS3" algorithm to compute the QR factorization of the columns of SYST(:m,:n), which are forced to be included in the basis as specified with the KFIX argument. However, if column pivoting is requested, QRFAC uses a

standard "BLAS2" algorithm without any blocking on the columns which must be pivoted and is thus not optimized for computing a full QR factorization with column pivoting of very large matrices.

QRFAC is an updated version of a subroutine with the same name provided in the reference (1) with improvements suggested in the reference (3).

For further details, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Drmac, Z., and Bujanovic, Z., 2006:** On the failure of rank revealing QR factorization software - a case study LAPACK Working Note 176.

6.16.8 subroutine qr_cmp2 (mat, diagr, beta, ip, krank, tol, tau)

Purpose

QR_CMP2 computes a (complete) orthogonal factorization of a real m-by-n matrix MAT. MAT may be rank-deficient. The routine first computes a QR factorization with column pivoting of MAT:

MAT * P = Q * R

, here P is an n-by-n permutation matrix, R is an upper triangular or trapezoidal (if n>m) matrix and Q is a m-by-m orthogonal matrix.

R can then be partioned by defining R11 as the largest leading submatrix of R whose estimated condition number, in the 1-norm, is less than 1/TOL (or such that abs(R[j,j])>0 if TOL is absent). The order of R11, krank, is the effective rank of MAT.

This leads implicitly to the following partition of Q:

[Q1 Q2]

where Q1 is an m-by-krank orthonormal matrix and Q2 is m-by-(m-krank) orthonormal matrix orthogonal to Q1, and to the following corresponding partition of R:

[R11 R12]

[R21 R22]

where R11 is a krank-by-krank triangular matrix, R21 is zero by construction, R12 is a full krank-by-(n-krank) matrix and R22 is a full (m-krank)-by-(n-krank) matrix.

Q1 and Q2 are, respectively, orthonormal bases of the range and null space of MAT.

In a final step, if TAU is present, R22 is considered to be negligible and R12 is annihilated by orthogonal transformations from the right, arriving at the complete orthogonal factorization:

MAT * P = Q * T * Z

, where P is a n-by-n permutation matrix, Q is a m-by-m orthogonal matrix, Z is a n-by-n orthogonal matrix and T is a m-by-n matrix and has the form:

[T11 T12]

[T21 T22]

Here T21 (=R21) and T12 are all zero, T22 (=R22) is considered to be negligible and T11 is a krank-by-krank upper triangular matrix.

On exit, P is stored compactly in the vector argument IP and if:

- TAU is absent, QR_CMP2 computes Q and submatrices R11, R12. Submatrices R11 and R12 are stored in the array arguments MAT and DIAGR. Q is stored compactly in factored form in the array arguments MAT and BETA.
- TAU is present, QR_CMP2 computes Q, Z and submatrice T11. Submatrice T11 is stored in the array arguments MAT and DIAGR. Q is stored compactly in factored form in the array arguments MAT and BETA. Z is stored compactly in factored form in the array arguments MAT and TAU.

See Further Details for more information on how the partial QR or orthogonal decomposition is stored in MAT on exit.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real m-by-n matrix to be decomposed.

On exit, MAT has been overwritten by details of its (complete) orthogonal factorization.

See Further Details.

DIAGR (OUTPUT) real(stnd), dimension(:) On exit, the diagonal elements of the matrix R if TAU is absent or the diagonal elements of the matrix T11 if TAU is present.

The diagonal elements of R or T11 are stored in DIAGR(1:krank) and krank is stored in the real argument KRANK on exit.

See Further Details.

The size of DIAGR must be equal to min(size(MAT,1) , size(MAT,2)).

BETA (OUTPUT) real(stnd), dimension(:) On exit, the scalars factors of the elementary reflectors defining Q.

See Further Details.

The size of BETA must be equal to min(size(MAT,1) , size(MAT,2)).

IP (OUTPUT) integer(i4b), dimension(:) On exit, if IP(j)=k, then the j-th column of MAT*P was the k-th column of MAT.

See Further Details.

The size of IP must be equal to size(MAT, 2) = n.

KRANK (INPUT/OUTPUT) integer(i4b) On entry:

- KRANK=k, implies that the first k columns of MAT are to be forced into the basis. Pivoting is performed on the last n-k columns of MAT.
- KRANK<=0 can be used when pivoting is desired on all columns of MAT.
- When KRANK>=min(m,n) is used, pivoting is not performed at all. This is appropriate when MAT is known to be full rank.

On exit, KRANK contains the effective rank of MAT, i.e., the order of the submatrix R11, krank. This is the same as the order of the submatrix T11 in the complete orthogonal factorization of MAT.

TOL (INPUT/OUTPUT, OPTIONAL) real(stnd) If TOL is present and is in [0,1[, then :

- On entry, the calculations to determine the condition number of MAT are performed. Then, TOL is used to determine the effective rank of MAT, which is defined as the order of the largest leading triangular submatrix R11 in the QR factorization with pivoting of MAT, whose estimated condition number < 1/TOL. If TOL=0 is specified the numerical rank of MAT is determined.
- On exit, the reciprocal of the condition number is returned in TOL.

If TOL is not specified or is outside [0,1[:

- The calculations to determine the condition number of MAT are not performed and crude tests on R(j,j) are done to determine the numerical rank of MAT. If TOL is present, it is not changed.

TAU (OUTPUT, OPTIONAL) real(stnd), dimension(:) On entry, if TAU is present, a complete orthogonal factorization of MAT is computed. Otherwise, a QR factorization with column pivoting of MAT is computed.

On exit, the scalars factors of the elementary reflectors defining the orthogonal matrix Z in the complete orthogonal factorization of MAT.

See Further Details.

The size of TAU must be equal to min(size(MAT,1) , size(MAT,2)).

Further Details

1) If it is possible that MAT may not be of full rank (i.e., certain columns of MAT are linear combinations of other columns), then the linearly dependent columns can usually be determined by using KRANK=0 and TOL=relative precision of the elements in MAT. If each element is correct to, say, 5 digits then TOL=0.00001 should be used. Also, it may be helpful to scale the columns of MAT so that all elements are about the same order of magnitude.

2) The matrix Q is represented as a product of elementary reflectors

   Q = H(1) * H(2) * . . . * H(k), where k = min(size(MAT,1) , size(MAT,2))

   Each H(i) has the form

   H(i) = I + beta * (v * v') ,

   where beta is a real scalar and v is a real m-element vector with v(1:i-1) = 0. v(i:m) is stored on exit in MAT(i:m,i) and beta in BETA(i).

   On exit of QR_CMP2, the orthonormal matrix Q (or its first n columns) can be computed explicitly by a call to subroutine ORTHO_GEN_QR with arguments MAT and BETA.

3) The matrix P is represented in the array IP as follows: If IP(j) = i then the jth column of P is the ith canonical unit vector.

4) If TAU is absent, a QR factorization with column pivoting of MAT is computed and, on exit:

- The elements above the diagonal of the array MAT(:krank,:krank) contain the corresponding elements of the triangular matrix R11.
- The elements of the diagonal of R11 are stored in the array DIAGR.
- The submatrix R12 is stored in MAT(:krank,krank+1:n).
- krank is stored in the real argument KRANK.

5) If TAU is present, a complete orthogonal factorization of MAT is computed. The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), which is used to introduce zeros into the kth row of R (e.g., in R12), is given in the form

   [I 0]

[0 T(k)]

where

T(k) = I + tau * (u(k) * u(k)') and u(k)' = (1 0 z(k))

tau is a scalar, u(k) is a n-k+1 vector and z(k) is an (n-krank) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of R12.

The Z n-by-n orthogonal matrix is given by

Z = Z(1) * Z(2) * ... * Z(krank)

On exit, the scalar tau is returned in the kth element of TAU and the vector u(k) in the kth row of MAT, such that the elements of z(k) are in MAT(k,krank+1:n). On exit:

- The scalar tau defining T(k) is returned in the kth element of TAU and the vector u(k) in the kth row of MAT, such that the elements of z(k) are in MAT(k,krank+1:n). The other elements of u(k) are not stored.
- The elements above the diagonal of the array section MAT(:krank,:krank) contain the corresponding elements of the triangular matrix T11. The elements of the diagonal of T11 are stored in the array DIAGR.
- krank is stored in the real argument KRANK.

The computations are parallelized if OPENMP is used. QR_CMP2 uses a blocked "BLAS3" algorithm to compute the QR factorization of the columns of MAT, which are forced to be included in the basis as specified with the KRANK argument on entry. However, if column pivoting is requested, QR_CMP2 uses a standard "BLAS2" algorithm without any blocking on the columns which must be pivoted and is thus not optimized for computing a full QR factorization with column pivoting of very large matrices.

For computing (partial) QR factorization with column pivoting of very large matrices, subroutines PARTIAL_RQR_CMP and PARTIAL_RQR_CMP2 in module Random are a better choice. These two subroutines are much faster than QR_CMP2 on large matrices because of the use of randomized and blocked "BLAS3" algorithms instead of a standard "BLAS2" algorithm as in QR_CMP2. See references (4), (5) and (6) for further information on randomized QR algorithms.

For further details, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Drmac, Z., and Bujanovic, Z., 2006:** On the failure of rank revealing QR factorization software - a case study LAPACK Working Note 176.

(4) **Duersch, J.A., and Gu, M., 2017:** Randomized QR with column pivoting. SIAM J. Sci. Comput., Volume 39, Issue 4, C263-C291.

(5) **Martinsson, P.G., Quintana-Orti, G., Heavner, N., and Van de Geijn, R., 2017:** Householder QR factorization with randomization for column pivoting (HQRRP). SIAM J. Sci. Comput., Volume 39, Issue 2, C96-C115.

(6) **Duersch, J.A., and Gu, M., 2020:** Randomized projection for rank-revealing matrix factorizations and low-rank approximations. SIAM Review, Volume 62, Issue 3, 661-682.

6.16.9 subroutine partial_qr_cmp (mat, diagr, beta, ip, krank, tol, tau)

Purpose

PARTIAL_QR_CMP computes a (partial) QR factorization with column pivoting or complete orthogonal factorization of a m-by-n matrix MAT:

MAT * P = Q * R

, where P is a n-by-n permutation matrix, R is an upper triangular or trapezoidal (i.e., if n>m) matrix and Q is a m-by-m orthogonal matrix.

At the user option, the QR factorization can be only partial, e.g., the subroutine ends when the numbers of selected columns of Q is equal to a predefined value equals to kpartial = size(DIAGR) = size(BETA).

This leads implicitly to the following partition of Q:

[Q1 Q2]

where Q1 is a m-by-kpartial orthonormal matrix and Q2 is a m-by-(m-kpartial) orthonormal matrix orthogonal to Q1, and to the following corresponding partition of R:

[R11 R12]

[R21 R22]

where R11 is a kpartial-by-kpartial triangular matrix, R21 is zero by construction, R12 is a full kpartial-by-(n-kpartial) matrix and R22 is a full (m-kpartial)-by-(n-kpartial) matrix.

Then, if the optional scalar argument TOL is present and:

- is in]0,1[, the rank of R11 is determined by finding the submatrix of R11 which is defined as the largest leading submatrix whose estimated condition number, in the 1-norm, is less than 1/TOL. The order of this submatrix, krank, is the effective rank of R11 (and MAT if krank is less than kpartial or if krank=kpartial=min(m,n)).
- is equal to 0, the rank of R11, krank, is determined by finding the largest submatrix of R11 such that abs(R11[j,j])>0.

In both cases, the order of this submatrix, krank, is the effective rank of R11 (and MAT if krank is less than kpartial or if krank=kpartial=min(m,n)).

If TOL is absent or outside [0,1[, the rank of R11 is not checked and is assumed to be equal to kpartial.

If krank is less than kpartial, then MAT is not of full rank (i.e., certain columns of MAT(:m,:kpartial) are linear combinations of other columns of MAT(:m,:kpartial)) and krank is also an estimate of the rank of MAT.

This leads to a redefinition of the partition of Q = [Q1 Q2], where Q1 and Q2 are now m-by-krank and m-by-(m-krank) orthonormal matrices, and a corresponding redefinition of the associated partition of R, where R11 is now a krank-by-krank triangular matrix, R21 is again zero by construction, R12 is a full krank-by-(n-krank) matrix and R22 is a (m-krank)-by-(n-krank) matrix.

In a final step, if TAU is present, R22 is considered to be negligible and R12 is annihilated by orthogonal transformations from the right, arriving at the partial or complete orthogonal factorization:

MAT * P = Q * T * Z

, where P is a n-by-n permutation matrix, Q is a m-by-m orthogonal matrix, Z is a n-by-n orthogonal matrix and T is a m-by-n matrix and has the form:

[T11 T12]

[T21 T22]

Here T21 (=R21) and T12 are all zero, T22 (=R22) is considered to be negligible and T11 is a krank-by-krank upper triangular matrix.

On exit, P is stored compactly in the vector argument IP, krank is stored in the scalar argument KRANK and if:

- TAU is absent, PARTIAL_QR_CMP computes Q and submatrices R11, R12 and R22. Submatrices R11, R12 and R22 are stored in the array arguments MAT and DIAGR. Q is stored compactly in factored form in the array arguments MAT and BETA.
- TAU is present, PARTIAL_QR_CMP computes Q, Z and submatrices T11 and T22 (=R22). Submatrices T11 and T22 are stored in the array arguments MAT and DIAGR. Q is stored compactly in factored form in the array arguments MAT and BETA. Z is stored compactly in factored form in the array arguments MAT and TAU.

See Further Details for more information.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real m-by-n matrix to be decomposed.

On exit, MAT has been overwritten by details of its (partial) QR factorization.

See Further Details.

DIAGR (OUTPUT) real(stnd), dimension(:) On exit, the diagonal elements of the matrix R11 or T11.

See Further Details.

The size of DIAGR must verify:

- size(DIAGR) = kpartial <= min(size(MAT,1) , size(MAT,2)).

BETA (OUTPUT) real(stnd), dimension(:) On exit, the scalars factors of the elementary reflectors defining Q.

See Further Details.

The size of BETA must verify:

- size(BETA) = kpartial <= min(size(MAT,1) , size(MAT,2)).

IP (OUTPUT) integer(i4b), dimension(:) On exit, if IP(j)=k, then the j-th column of MAT*P was the k-th column of MAT.

See Further Details.

The size of IP must be equal to size(MAT, 2) = n.

KRANK (OUTPUT) integer(i4b) On exit, KRANK contains the effective rank of R11, i.e., krank, which is the order of this submatrix R11. This is the same as the order of the submatrix T11 in the complete orthogonal factorization of MAT and is also the rank of MAT if krank is less than kpartial = size(BETA).

If the computed pseudo-rank, krank, is less than kpartial = size(BETA), BETA(krank+1:kpartial) and, eventually, TAU(krank+1:kpartial) are set to zero and MAT(krank+1:m,krank+1:n) (e.g., R22=T22) is updated on exit. In other words, the subroutine outputs a partial QR factorization of rank krank instead of rank kpartial.

In all cases, norm(MAT(krank+1:m,krank+1:n)) gives the error of the associated matrix approximation in the Frobenius norm, on exit.

TOL (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, if TOL is present then:

- if TOL is in]0,1[, the calculations to determine the condition number of R11 are performed. Then, TOL is used to determine the effective pseudo-rank of R11, which is defined as the order of the largest leading triangular submatrix in the partial QR factorization with column pivoting of MAT, whose estimated condition number in the 1-norm is less than 1/TOL. On exit, the reciprocal of the condition number is returned in TOL.
- if TOL=0 is specified, the calculations to determine the condition number of R11 are not performed and crude tests on R(j,j) are done to determine the numerical pseudo-rank of R11. On exit, TOL is not changed.

If TOL is not specified or is outside [0,1[, the calculations to determine the rank of R11 are not performed and this rank is assumed to be equal to kpartial.

TAU (OUTPUT, OPTIONAL) real(stnd), dimension(:) On entry, if TAU is present, a (partial) complete orthogonal factorization of MAT is computed. Otherwise, a simple QR factorization with column pivoting of MAT is computed.

On exit, the scalars factors of the elementary reflectors defining the orthogonal matrix Z in the (partial) complete orthogonal factorization of MAT.

See Further Details.

The size of TAU must verify:

- size(TAU) = kpartial <= min(size(MAT,1) , size(MAT,2)).

Further Details

The matrix Q is represented as a product of elementary reflectors

Q = H(1) * H(2) * . . . * H(krank), where krank = size(BETA) <= min(m , n).

Each H(i) has the form

H(i) = I + beta * (v * v') ,

where beta is a real scalar and v is a real m-element vector with v(1:i-1) = 0. v(i:m) is stored on exit in MAT(i:m,i) and beta in BETA(i).

On exit of PARTIAL_QR_CMP, the orthonormal matrix Q (or its first n columns) can be computed explicitly by a call to subroutine ORTHO_GEN_QR with arguments MAT and BETA.

The matrix P is represented in the array IP as follows: If IP(j) = i then the jth column of P is the ith canonical unit vector.

On exit, if the optional argument TAU is absent:

- The elements above the diagonal of the array MAT(:krank,:krank) contain the corresponding elements of the triangular matrix R11.
- The elements of the diagonal of R11 are stored in the array DIAGR.
- The submatrix R12 is stored in MAT(:krank,krank+1:n).
- The submatrix R22 is stored in MAT(krank+1:m,krank+1:n).
- krank is stored in the real argument KRANK.

If TAU is present, a partial or complete orthogonal factorization of MAT is computed. The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), which is used to introduce zeros into the kth row of R (e.g., in R12), is given in the form

[I 0]

[0 T(k)]

where

T(k) = I + tau * (u(k) * u(k)') and u(k)' = (1 0 z(k))

tau is a scalar, u(k) is a n-k+1 vector and z(k) is an (n-krank) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of R12.

The Z n-by-n orthogonal matrix is given by

Z = Z(1) * Z(2) * ... * Z(krank)

On exit, if the optional argument TAU is present:

- The scalar tau defining T(k) is returned in the kth element of TAU and the vector u(k) in the kth row of MAT, such that the elements of z(k) are in MAT(k,krank+1:n). The other elements of u(k) are not stored.
- The elements above the diagonal of the array section MAT(:krank,:krank) contain the corresponding elements of the triangular matrix T11. The elements of the diagonal of T11 are stored in the array DIAGR.
- The submatrix T22 (=R22) is stored in MAT(krank+1:m,krank+1:n).
- krank is stored in the real argument KRANK.

In both cases, norm(MAT(krank+1:m,krank+1:n)) gives the error of the associated matrix approximation in the Frobenius norm, on exit.

If it is possible that MAT may not be of full rank (i.e., certain columns of MAT are linear combinations of other columns), then the eventual linearly dependent columns in the partial QR decomposition of MAT, which is sought, can be determined by using TOL=relative precision of the elements in MAT and the partial QR or complete orthogonal factorization of MAT is adjusted accordingly. If each element is correct to, say, 5 digits then TOL=0.00001 should be used. Also, it may be helpful to scale the columns of MAT so that all elements are about the same order of magnitude.

The computations are parallelized if OPENMP is used. However, note that PARTIAL_QR_CMP uses a standard "BLAS2" algorithm without any blocking and is thus not optimized for computing a partial QR or complete orthogonal factorization with column pivoting of very large matrices. For large matrices, subroutines PARTIAL_RQR_CMP and PARTIAL_RQR_CMP2 in module Random, which use randomized blocked "BLAS3" algorithms described in the references (4), (5) and (6), are a much better choice.

For further details, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Drmac, Z., and Bujanovic, Z., 2006:** On the failure of rank revealing QR factorization software - a case study LAPACK Working Note 176.

(4) **Duersch, J.A., and Gu, M., 2017:** Randomized QR with column pivoting. SIAM J. Sci. Comput., Volume 39, Issue 4, C263-C291.

(5) **Martinsson, P.G., Quintana-Orti, G., Heavner, N., and Van de Geijn, R., 2017:** Householder QR factorization with randomization for column pivoting (HQRRP). SIAM J. Sci. Comput., Volume 39, Issue 2, C96-C115.

(6) **Duersch, J.A., and Gu, M., 2020:** Randomized projection for rank-revealing matrix factorizations and low-rank approximations. SIAM Review, Volume 62, Issue 3, 661-682.

6.16.10 subroutine partial_qr_cmp_fixed_precision (mat, relerr, diagr, beta, ip, krank, tau)

Purpose

PARTIAL_QR_CMP_FIXED_PRECISION computes a partial QR factorization with column pivoting (or orthogonal factorization) of a m-by-n matrix MAT:

MAT * P = Q * R

, where P is a n-by-n permutation matrix, R is a krank-by-n (upper trapezoidal) matrix and Q is a m-by-krank matrix with orthogonal columns. This leads to the following matrix approximation of MAT of rank krank:

MAT = Q * (R * P')

krank is the target rank of the matrix approximation, which is sought, and this partial factorization must have an approximation error which fulfills:

|| MAT - Q * (R * P') ||_F <= ||MAT||_F * relerr

|| ||_F is the Frobenius norm and relerr is a prescribed accuracy tolerance for the relative error of the computed matrix approximation, specified in the input argument RELERR.

PARTIAL_QR_CMP_FIXED_PRECISION searches incrementally the best (e.g., of smallest rank) Q * (R * P') approximation, which fulfills the prescribed accuracy tolerance for the relative error. More precisely, the rank of the matrix approximation is increased progressively until the prescribed accuracy tolerance is satisfied.

In other words, the rank, krank, of the matrix approximation is not known in advance and is determined in the subroutine. krank is stored in the argument KRANK and the relative error of the computed partial matrix approximation in the argument RELERR on exit.

The computed partial matrix approximation leads implicitly to the following partition of R:

[R1 R2]

where R1 is a krank-by-krank triangular matrix and R2 is a full krank-by-(n-krank) matrix.

In a final step, if TAU is present, R2 is annihilated by orthogonal transformations from the right, arriving at the partial orthogonal factorization:

MAT * P = Q * T1 * Z

, where P is a n-by-n permutation matrix, Q is a m-by-krank matrix with orthonormal columns Z is a krank-by-n matrix with orthonormal rows and T1 is a krank-by-krank upper triangular matrix.

Note, however, that this final step does not change the matrix approximation and its relative error, only the output format of this matrix approximation, which is now composed of four factors instead of three.

On exit, P is stored compactly in the vector argument IP, krank is stored in the scalar argument KRANK and if:

- TAU is absent, PARTIAL_QR_CMP_FIXED_PRECISION computes Q and submatrices R1 and R2. Submatrices R1 and R2 are stored in the array arguments MAT and DIAGR(:KRANK). Q is stored compactly in factored form in the array arguments MAT and BETA(:KRANK).
- TAU is present, PARTIAL_QR_CMP_FIXED_PRECISION computes Q, Z and submatrice T1. Submatrice T1 is stored in the array arguments MAT and DIAGR. Q is stored compactly in factored form in the array arguments MAT and BETA(:KRANK). Z is stored compactly in factored form in the array arguments MAT and TAU(:KRANK).

In all cases, the relative error of the computed matrix approximation is output in argument RELERR.

See Further Details for more information.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real m-by-n matrix to be decomposed.

On exit, MAT has been overwritten by details of its partial QR or complete orthogonal factorization.

See Further Details.

RELERR (INPUT/OUTPUT) real(stnd) On entry, the requested accuracy tolerance for the relative error of the computed partial matrix approximation.

The preset value for RELERR must be greater than 4*epsilon(RELERR) and less than one.

On exit, RELERR contains the relative error of the computed partial matrix approximation:

- RELERR = || MAT - Q * (R * P') ||_F / ||MAT||_F

DIAGR (OUTPUT) real(stnd), dimension(:) On exit, the diagonal elements of the matrix R1 or T1 are stored in the array section DIAGR(:KRANK). Other elements of DIAGR are set to zero on exit.

See Further Details.

The size of DIAGR must verify:

- size(DIAGR) = min(size(MAT,1) , size(MAT,2)).

BETA (OUTPUT) real(stnd), dimension(:) On exit, the scalars factors of the elementary reflectors defining Q are stored in the array section BETA(:KRANK). Other elements of BETA are set to zero on exit.

See Further Details.

The size of BETA must verify:

- size(BETA) = min(size(MAT,1) , size(MAT,2)).

IP (OUTPUT) integer(i4b), dimension(:) On exit, if IP(j)=k, then the j-th column of MAT*P was the k-th column of MAT.

See Further Details.

The size of IP must be equal to size(MAT, 2) = n.

KRANK (OUTPUT) integer(i4b) On exit, KRANK contains the rank of R1, i.e., krank, which is the order of this submatrix R1. This is the same as the order of the submatrix T1 in the "partial" complete orthogonal factorization of MAT and is also the rank of the computed matrix approximation.

In all cases, norm(MAT(KRANK+1:m,KRANK+1:n)) gives the error of the associated matrix approximation in the Frobenius norm, on exit.

TAU (OUTPUT, OPTIONAL) real(stnd), dimension(:) On entry, if TAU is present, a partial complete orthogonal factorization of MAT is computed. Otherwise, a simple QR factorization with column pivoting of MAT is computed.

On exit, the scalars factors of the elementary reflectors defining the orthogonal matrix Z in the partial complete orthogonal factorization of MAT are stored in the array section TAU(:KRANK). Other elements of TAU are set to zero on exit.

See Further Details.

The size of TAU must verify:

- size(TAU) = min(size(MAT,1) , size(MAT,2)).

Further Details

The matrix Q is represented as a product of elementary reflectors

Q = H(1) * H(2) * . . . * H(krank), where krank <= min(m , n).

Each H(i) has the form

H(i) = I + beta * (v * v') ,

where beta is a real scalar and v is a real m-element vector with v(1:i-1) = 0. v(i:m) is stored on exit in MAT(i:m,i) and beta in BETA(i).

On exit of PARTIAL_QR_CMP_FIXED_PRECISION, the matrix Q can be computed explicitly by a call to subroutine ORTHO_GEN_QR with arguments MAT and BETA(:KRANK).

The matrix P is represented in the array IP as follows: If IP(j) = i then the jth column of P is the ith canonical unit vector.

On exit, if the optional argument TAU is absent:

- The elements above the diagonal of the array MAT(:krank,:krank) contain the corresponding elements of the triangular matrix R1.
- The elements of the diagonal of R1 are stored in the array DIAGR.
- The submatrix R2 is stored in MAT(:krank,krank+1:n).
- krank is stored in the real argument KRANK.

If TAU is present, a "partial" complete orthogonal factorization of MAT is computed. The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), which is used to introduce zeros into the kth row of R (e.g., in R2), is given in the form

[I 0]

[0 T(k)]

where

T(k) = I + tau * (u(k) * u(k)') and u(k)' = (1 0 z(k))

tau is a scalar, u(k) is a n-k+1 vector and z(k) is an (n-krank) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of R2.

The Z n-by-n orthogonal matrix is given by

Z = Z(1) * Z(2) * . . . * Z(krank)

On exit, if the optional argument TAU is present:

- The scalar tau defining T(k) is returned in the kth element of TAU and the vector u(k) in the kth row of MAT, such that the elements of z(k) are in MAT(k,krank+1:n). The other elements of u(k) are not stored.
- The elements above the diagonal of the array section MAT(:krank,:krank) contain the corresponding elements of the triangular matrix T1. The elements of the diagonal of T1 are stored in the array DIAGR.
- krank is stored in the real argument KRANK.

In both cases, norm(MAT(KRANK+1:m,KRANK+1:n)) gives the error of the associated partial matrix approximation in the Frobenius norm, and argument RELERR stores the relative error in the Frobenius norm of the matrix approximation on exit.

The computations are parallelized if OPENMP is used. However, note that PARTIAL_QR_CMP_FIXED_PRECISION uses a standard "BLAS2" algorithm without any blocking and is thus not optimized for computing a partial QR or complete orthogonal factorization with column pivoting of very large matrices. For large matrices, subroutine PARTIAL_RQR_CMP_FIXED_PRECISION in module Random, which uses a randomized blocked "BLAS3" algorithm described in the references (4), (5) and (6) is a much better choice.

For further details, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Drmac, Z., and Bujanovic, Z., 2006:** On the failure of rank revealing QR factorization software - a case study LAPACK Working Note 176.

(4) **Duersch, J.A., and Gu, M., 2017:** Randomized QR with column pivoting. SIAM J. Sci. Comput., Volume 39, Issue 4, C263-C291.

(5) **Martinsson, P.G., Quintana-Orti, G., Heavner, N., and Van de Geijn, R., 2017:** Householder QR factorization with randomization for column pivoting (HQRRP). SIAM J. Sci. Comput., Volume 39, Issue 2, C96-C115.

(6) **Duersch, J.A., and Gu, M., 2020:** Randomized projection for rank-revealing matrix factorizations and low-rank approximations. SIAM Review, Volume 62, Issue 3, 661-682.

6.16.11 `subroutine ortho_gen_qr ( mat, beta )`

Purpose

ORTHO_GEN_QR generates an m-by-n real matrix with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m

Q = H(1) * H(2) * . . . * H(k)

as returned by QR_CMP, QR_CMP_TS or QR_CMP2.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,. . . ,k, as returned by QR_CMP (or QR_CMP_TS or QR_CMP2) in the first k columns of its array argument MAT.

On exit, the first n columns of Q.

The shape of MAT must verify: size(MAT, 2) <= size(MAT, 1).

BETA (INPUT) real(stnd), dimension(:) BETA(i) must contain the scalar factor of the elementary reflector H(i), as returned by QR_CMP (or QR_CMP_TS or QR_CMP2). The size of BETA determines the number k of elementary reflectors whose product defines the matrix Q.

The size of BETA must verify: size(BETA) <= (MAT, 2) .

Further Details

This subroutine used a blocked algorithm for agregating the Householder transformations (e.g. the elementary reflectors) stored in the lower triangle of MAT and generating the orthogonal matrix Q of the QR factorization.

Furthermore, the computations are parallelized if OPENMP is used.

For further details on the QR factorization and its use or the blocked algorithm, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(4) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.16.12 `subroutine ortho_gen_qr_ts ( mat, beta )`

Purpose

ORTHO_GEN_QR_TS generates an m-by-n real matrix with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m

Q = H(1) * H(2) * . . . * H(k)

as returned by QR_CMP, QR_CMP_TS or QR_CMP2.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,. . . ,k, as returned by QR_CMP (or QR_CMP_TS or QR_CMP2) in the first k columns of its array argument MAT.

On exit, the first n columns of Q.

The shape of MAT must verify: size(MAT, 2) <= size(MAT, 1).

BETA (INPUT) real(stnd), dimension(:) BETA(i) must contain the scalar factor of the elementary reflector H(i), as returned by QR_CMP (or QR_CMP_TS or QR_CMP2). The size of BETA determines the number k of elementary reflectors whose product defines the matrix Q.

The size of BETA must verify: size(BETA) <= (MAT, 2) .

Further Details

This subroutine used a blocked algorithm for agregating the Householder transformations (e.g. the elementary reflectors) stored in the lower triangle of MAT and generating the orthogonal matrix Q of the QR factorization.

Furthermore, the computations are parallelized if OPENMP is used.

This subroutine performs exactly the same task as subroutine ORTHO_GEN_QR, but will be sometime more robust (and efficient) when OpenMP is used, for huge tall and skinny m-by-n matrices MAT, as ORTHO_GEN_QR_TS uses only BLAS1 kernels instead of BLAS3 and BLAS2 kernels as in ORTHO_GEN_QR in the blocked algorithm.

For further details on the QR factorization and its use or the blocked algorithm, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(4) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.16.13 `subroutine apply_q_qr ( mat, beta, c, left, trans )`

Purpose

APPLY_Q_QR overwrites the general real m-by-n matrix C with

Q * C if LEFT = true and TRANS = false, or

Q' * C if LEFT = true and TRANS = true, or

C * Q if LEFT = false and TRANS = false, or

C * Q' if LEFT = false and TRANS = true,

where Q is a real orthogonal matrix defined as the product of k elementary reflectors

Q = H(1) * H(2) * ... * H(k)

as returned by QR_CMP or QR_CMP2. Q is of order m if LEFT = true and of order n if LEFT = false.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,...,k, as returned by QR_CMP (or QR_CMP2) in the first k columns of its array argument MAT. MAT is not modified by the routine.

BETA (INPUT) real(stnd), dimension(:) BETA(i) must contain the scalar factor of the elementary reflector H(i), as returned by QR_CMP (or QR_CMP2). The size of BETA determines the number k of elementary reflectors whose product defines the matrix Q.

The size of BETA must verify:

- size(BETA) <= min(size(MAT,1) , size(MAT,2)) .

C (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m by n matrix C.

On exit, C is overwritten by Q * C or Q' * C or C * Q' or C * Q.

The shape of C must verify:

- if LEFT = true, size(C, 1) = size(MAT, 1)
- if LEFT = false, size(C, 2) = size(MAT, 1) .

LEFT (INPUT) logical(lgl) If:

- LEFT = true : apply Q or Q' from the left
- LEFT = false : apply Q or Q' from the right

TRANS (INPUT) logical(lgl) If:

- TRANS = false : apply Q (no transpose)
- TRANS = true : apply Q' (transpose)

Further Details

This subroutine is adapted from the routine DORM2R in LAPACK.

This subroutine used a blocked algorithm for agregating the Householder transformations (e.g. the elementary reflectors) stored in the lower triangle of MAT and applying the orthogonal matrix Q of the QR factorization to the real m-by-n matrix C.

Furthermore, the computations are parallelized if OPENMP is used.

For further details on the QR factorization and its use or the blocked algorithm used here, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(4) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.16.14 `subroutine apply_q_qr ( mat, beta, c, trans )`

Purpose

APPLY_Q_QR overwrites the real m vector C with

Q * C if TRANS = false, or

Q' * C if TRANS = true

where Q is a real orthogonal matrix defined as the product of k elementary reflectors

Q = H(1) * H(2) * . . . * H(k)

as returned by QR_CMP or QR_CMP2. Q is of order m .

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,. . . ,k, as returned by QR_CMP (or QR_CMP2) in the first k columns of its array argument MAT. MAT is not modified by the routine.

BETA (INPUT) real(stnd), dimension(:) BETA(i) must contain the scalar factor of the elementary reflector H(i), as returned by QR_CMP (or QR_CMP2). The size of BETA determines the number k of elementary reflectors whose product defines the matrix Q.

The size of BETA must verify: size(BETA) <= min(size(MAT,1) , size(MAT,2)) .

C (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the m vector C.

On exit, C is overwritten by Q * C or Q' * C .

The shape of C must verify: size(C) = size(MAT, 1) .

TRANS (INPUT) logical(lgl) If:

- TRANS = false : apply Q (no transpose)
- TRANS = true : apply Q' (transpose)

Further Details

This subroutine is adapted from the routine DORM2R in LAPACK.

For further details on the QR factorization and its use or the algorithm used here, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.16.15 `subroutine apply_q_qr_ts ( mat, beta, c, left, trans )`

Purpose

APPLY_Q_QR_TS overwrites the general real m-by-n matrix C with

Q * C if LEFT = true and TRANS = false, or

Q' * C if LEFT = true and TRANS = true, or

C * Q if LEFT = false and TRANS = false, or

C * Q' if LEFT = false and TRANS = true,

where Q is a real orthogonal matrix defined as the product of k elementary reflectors

Q = H(1) * H(2) * . . . * H(k)

as returned by QR_CMP or QR_CMP2. Q is of order m if LEFT = true and of order n if LEFT = false.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,. . . ,k, as returned by QR_CMP (or QR_CMP2) in the first k columns of its array argument MAT. MAT is not modified by the routine.

BETA (INPUT) real(stnd), dimension(:) BETA(i) must contain the scalar factor of the elementary reflector H(i), as returned by QR_CMP (or QR_CMP2). The size of BETA determines the number k of elementary reflectors whose product defines the matrix Q.

The size of BETA must verify:

- size(BETA) <= min(size(MAT,1) , size(MAT,2)) .

C (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m by n matrix C.

On exit, C is overwritten by Q * C or Q' * C or C * Q' or C * Q.

The shape of C must verify:

- if LEFT = true, size(C, 1) = size(MAT, 1)
- if LEFT = false, size(C, 2) = size(MAT, 1) .

LEFT (INPUT) logical(lgl) If:

- LEFT = true : apply Q or Q' from the left
- LEFT = false : apply Q or Q' from the right

TRANS (INPUT) logical(lgl) If:

- TRANS = false : apply Q (no transpose)
- TRANS = true : apply Q' (transpose)

Further Details

This subroutine is adapted from the routine DORM2R in LAPACK.

This subroutine used a blocked algorithm for agregating the Householder transformations (e.g. the elementary reflectors) stored in the lower triangle of MAT and applying the orthogonal matrix Q of the QR factorization to the real m-by-n matrix C.

Furthermore, the computations are parallelized if OPENMP is used.

This subroutine performs exactly the same task as subroutine APPLY_Q_QR, but will be more efficient when OpenMP is used for tall and skinny m-by-n matrices C with a number of columns less than the module parameter BLKSZ if LEFT = true or for short and wide matrices C with a number of rows less than the module parameter BLKSZ if LEFT = false .

For further details on the QR factorization and its use or the blocked algorithm used here, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(4) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.16.16 `subroutine apply_q_qr_ts ( mat, beta, c, trans )`

Purpose

APPLY_Q_QR_TS overwrites the real m vector C with

Q * C if TRANS = false, or

Q' * C if TRANS = true

where Q is a real orthogonal matrix defined as the product of k elementary reflectors

Q = H(1) * H(2) * . . . * H(k)

as returned by QR_CMP or QR_CMP2. Q is of order m .

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,. . . ,k, as returned by QR_CMP (or QR_CMP2) in the first k columns of its array argument MAT. MAT is not modified by the routine.

BETA (INPUT) real(stnd), dimension(:) BETA(i) must contain the scalar factor of the elementary reflector H(i), as returned by QR_CMP (or QR_CMP2). The size of BETA determines the number k of elementary reflectors whose product defines the matrix Q.

The size of BETA must verify: size(BETA) <= min(size(MAT,1) , size(MAT,2)) .

C (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the m vector C.

On exit, C is overwritten by Q * C or Q' * C .

The shape of C must verify: size(C) = size(MAT, 1) .

TRANS (INPUT) logical(lgl) If:

- TRANS = false : apply Q (no transpose)
- TRANS = true : apply Q' (transpose)

Further Details

This subroutine is adapted from the routine DORM2R in LAPACK.

For further details on the QR factorization and its use or the algorithm used here, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.17 Module_Random

Copyright 2025 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

THIS MODULE REPLACES THE FORTRAN 90 INTRINSICS random_number AND random_seed BY SEVERAL IMPLEMENTATIONS OF THE KISS (Keep It Simple Stupid), L'Ecuyer's LFSR113, MERSENNE TWISTER MT19937 AND MEMT19937-II RANDOM NUMBER GENERATORS.

IN ADDITION TO 10 DIFFERENT UNIFORM RANDOM GENERATORS, GAUSSIAN RANDOM GENERATORS, SHUFFLING AND SAMPLING SUBROUTINES ARE ALSO PROVIDED, AS WELL AS SUBROUTINES FOR GENERATING PSEUDO-RANDOM ORTHOGONAL MATRICES FOLLOWING THE HAAR DISTRIBUTION OVER THE GROUP OF ORTHOGONAL MATRICES, PSEUDO_RANDOM SYMMETRIC MATRICES WITH A PRESCRIBED SPECTRUM OR PSEUDO_RANDOM MATRICES WITH A PRESCRIBED SINGULAR VALUE DISTRIBUTION. FINALLY, RANDOMIZED LINEAR ALGEBRA ROUTINES LIKE RANDOMIZED QB, QR, COLUMN INTERPOLATIVE, TWO_SIDED INTERPOLATIVE AND CUR DECOMPOSITIONS ARE ALSO INCLUDED.

MANY PARTS OF THIS MODULE ARE ADAPTED FROM :

Hennecke, M., 1995: A Fortran90 interface to random number generation. Computer Physics Communications. Volume 90, Number 1, 117-120

LATEST REVISION : 15/09/2025

6.17.1 `function rand_number ()`

purpose

This function returns a uniformly distributed random number between 0 and 1, exclusive of the two endpoints 0 and 1.

Arguments

None

Further Details

If the CPP macro _RANDOM_WITH0 is used during compilation, this routine may return 0 value.

6.17.2 `subroutine random_number_ ( harvest )`

purpose

This subroutine returns a uniformly distributed random number HARVEST between 0 and 1, exclusive of the two endpoints 0 and 1.

Arguments

HARVEST (OUTPUT) real(stnd) A uniformly distributed random real number between 0 and 1.

Further Details

If the CPP macro _RANDOM_WITH0 is used during compilation, this routine may return 0 value.

6.17.3 subroutine random_number_ (harvest)

purpose

This subroutine returns a uniformly distributed random vector HARVEST between 0 and 1, exclusive of the two endpoints 0 and 1.

Arguments

HARVEST (OUTPUT) real(stnd), dimension(:) A uniformly distributed random real vector between 0 and 1.

Further Details

If the CPP macro _RANDOM_WITH0 is used during compilation, this routine may return 0 value.

6.17.4 subroutine random_number_ (harvest)

purpose

This subroutine returns a uniformly distributed random matrix HARVEST between 0 and 1, exclusive of the two endpoints 0 and 1.

Arguments

HARVEST (OUTPUT) real(stnd), dimension(:,:) A uniformly distributed random real matrix between 0 and 1.

Further Details

If the CPP macro _RANDOM_WITH0 is used during compilation, this routine may return 0 value.

6.17.5 subroutine random_number_ (harvest)

purpose

This subroutine returns a uniformly distributed random array of dimension 3 HARVEST between 0 and 1, exclusive of the two endpoints 0 and 1.

Arguments

HARVEST (OUTPUT) real(stnd), dimension(:,:,:) A uniformly distributed random real array of dimension 3 between 0 and 1.

Further Details

If the CPP macro _RANDOM_WITH0 is used during compilation, this routine may return 0 value.

6.17.6 subroutine random_number_ (harvest)

purpose

This subroutine returns a uniformly distributed random array of dimension 4 HARVEST between 0 and 1, exclusive of the two endpoints 0 and 1.

Arguments

HARVEST (OUTPUT) real(stnd), dimension(:,:,:,:) A uniformly distributed random real array of dimension 4 between 0 and 1.

Further Details

If the CPP macro _RANDOM_WITH0 is used during compilation, this routine may return 0 value.

6.17.7 subroutine random_number_ (harvest)

purpose

This subroutine returns a uniformly distributed random array of dimension 5 HARVEST between 0 and 1, exclusive of the two endpoints 0 and 1.

Arguments

HARVEST (OUTPUT) real(stnd), dimension(:,:,:,:,:) A uniformly distributed random real array of dimension 5 between 0 and 1.

Further Details

If the CPP macro _RANDOM_WITH0 is used during compilation, this routine may return 0 value.

6.17.8 subroutine random_number_ (harvest)

purpose

This subroutine returns a uniformly distributed random array of dimension 6 HARVEST between 0 and 1, exclusive of the two endpoints 0 and 1.

Arguments

HARVEST (OUTPUT) real(stnd), dimension(:,:,:,:,:,:) A uniformly distributed random real array of dimension 6 between 0 and 1.

Further Details

If the CPP macro _RANDOM_WITH0 is used during compilation, this routine may return 0 value.

6.17.9 subroutine random_number_ (harvest)

purpose

This subroutine returns a uniformly distributed random array of dimension 7 HARVEST between 0 and 1, exclusive of the two endpoints 0 and 1.

Arguments

HARVEST (OUTPUT) real(stnd), dimension(:,:,:,:,:,:,:) A uniformly distributed random real array of dimension 7 between 0 and 1.

Further Details

If the CPP macro _RANDOM_WITH0 is used during compilation, this routine may return 0 value.

6.17.10 function rand_integer32 ()

purpose

This function returns a random integer in the interval (-2147483648,2147483647) inclusive of the two endpoints. The returned integer is equivalent to a signed 32-bit integer.

Arguments

None

6.17.11 subroutine random_integer32_ (harvest)

purpose

This subroutine returns a random integer in the interval (-2147483648,2147483647) inclusive of the two endpoints. The returned integer is equivalent to a signed 32-bit integer.

Arguments

HARVEST (OUTPUT) integer(i4b) A random integer in the interval (-2147483648,2147483647).

6.17.12 subroutine random_integer32_ (harvest)

purpose

This subroutine returns a vector of random integers in the interval (-2147483648,2147483647) inclusive of the two endpoints. The returned integers are equivalent to signed 32-bit integers.

Arguments

HARVEST (OUTPUT) integer(i4b), dimension(:) A vector of random integers in the interval (-2147483648,2147483647).

6.17.13 subroutine random_integer32_ (harvest)

purpose

This subroutine returns a matrix of random integers in the interval (-2147483648,2147483647) inclusive of the two endpoints. The returned integers are equivalent to signed 32-bit integers.

Arguments

HARVEST (OUTPUT) integer(i4b), dimension(:,:) A matrix of random integers in the interval (-2147483648,2147483647).

6.17.14 subroutine random_integer32_ (harvest)

purpose

This subroutine returns an array of random integers in the interval (-2147483648,2147483647) inclusive of the two endpoints. The returned integers are equivalent to signed 32-bit integers.

Arguments

HARVEST (OUTPUT) integer(i4b), dimension(:,:,:) An array of random integers in the interval (-2147483648,2147483647).

6.17.15 subroutine random_integer32_ (harvest)

purpose

This subroutine returns an array of random integers in the interval (-2147483648,2147483647) inclusive of the two endpoints. The returned integers are equivalent to signed 32-bit integers.

Arguments

HARVEST (OUTPUT) integer(i4b), dimension(:,:,:,:) An array of random integers in the interval (-2147483648,2147483647).

6.17.16 subroutine random_integer32_ (harvest)

purpose

This subroutine returns an array of random integers in the interval (-2147483648,2147483647) inclusive of the two endpoints. The returned integers are equivalent to signed 32-bit integers.

Arguments

HARVEST (OUTPUT) integer(i4b), dimension(:,:,:,:,:) An array of random integers in the interval (-2147483648,2147483647).

6.17.17 subroutine random_integer32_ (harvest)

purpose

This subroutine returns an array of random integers in the interval (-2147483648,2147483647) inclusive of the two endpoints. The returned integers are equivalent to signed 32-bit integers.

Arguments

HARVEST (OUTPUT) integer(i4b), dimension(:,:,:,:,:,:,:) An array of random integers in the interval (-2147483648,2147483647).

6.17.18 subroutine random_integer32_ (harvest)

purpose

This subroutine returns an array of random integers in the interval (-2147483648,2147483647) inclusive of the two endpoints. The returned integers are equivalent to signed 32-bit integers.

Arguments

HARVEST (OUTPUT) integer(i4b), dimension(:,:,:,:,:,:,:,:) An array of random integers in the interval (-2147483648,2147483647).

6.17.19 function rand_integer31 ()

purpose

This function returns a random integer in the interval (0,2147483647) inclusive of the two endpoints. The returned integer is equivalent to an unsigned 31-bit integer.

Arguments

None

6.17.20 subroutine random_integer31_ (harvest)

purpose

This subroutine returns a random integer in the interval (0,2147483647) inclusive of the two endpoints. The returned integer is equivalent to an unsigned 31-bit integer.

Arguments

HARVEST (OUTPUT) integer(i4b) A random integer in the interval (0,2147483647).

6.17.21 subroutine random_integer31_ (harvest)

purpose

This subroutine returns a vector of random integers in the interval (0,2147483647) inclusive of the two endpoints. The returned integers are equivalent to unsigned 31-bit integers.

Arguments

HARVEST (OUTPUT) integer(i4b), dimension(:) A vector of random integers in the interval (0,2147483647).

6.17.22 subroutine random_integer31_ (harvest)

purpose

This subroutine returns a matrix of random integers in the interval (0,2147483647) inclusive of the two endpoints. The returned integers are equivalent to unsigned 31-bit integers.

Arguments

HARVEST (OUTPUT) integer(i4b), dimension(:,:) A matrix of random integers in the interval (0,2147483647).

6.17.23 subroutine random_integer31_ (harvest)

purpose

This subroutine returns an array of random integers in the interval (0,2147483647) inclusive of the two endpoints. The returned integers are equivalent to unsigned 31-bit integers.

Arguments

HARVEST (OUTPUT) integer(i4b), dimension(:,:,:) An array of random integers in the interval (0,2147483647).

6.17.24 subroutine random_integer31_ (harvest)

purpose

This subroutine returns an array of random integers in the interval (0,2147483647) inclusive of the two endpoints. The returned integers are equivalent to unsigned 31-bit integers.

Arguments

HARVEST (OUTPUT) integer(i4b), dimension(:,:,:,:) An array of random integers in the interval (0,2147483647).

6.17.25 subroutine random_integer31_ (harvest)

purpose

This subroutine returns an array of random integers in the interval (0,2147483647) inclusive of the two endpoints. The returned integers are equivalent to unsigned 31-bit integers.

Arguments

HARVEST (OUTPUT) integer(i4b), dimension(:,:,:,:,:) An array of random integers in the interval (0,2147483647).

6.17.26 subroutine random_integer31_ (harvest)

purpose

This subroutine returns an array of random integers in the interval (0,2147483647) inclusive of the two endpoints. The returned integers are equivalent to unsigned 31-bit integers.

Arguments

HARVEST (OUTPUT) integer(i4b), dimension(:,:,:,:,:,:) An array of random integers in the interval (0,2147483647).

6.17.27 subroutine random_integer31_ (harvest)

purpose

This subroutine returns an array of random integers in the interval (0,2147483647) inclusive of the two endpoints. The returned integers are equivalent to unsigned 31-bit integers.

Arguments

HARVEST (OUTPUT) integer(i4b), dimension(:,:,:,:,:,:,:) An array of random integers in the interval (0,2147483647).

6.17.28 subroutine init_mt19937 (seed)

purpose

User interface subroutine for initializing the state of the MT19937 Random Number Generator (RNG) with a scalar seed, directly, without using the subroutine RANDOM_SEED_ and its interface.

Arguments

SEED (INPUT) integer(i4b) On entry, a scalar integer that will be used to initialize the MT19937 RNG.

Further Details

Only the first 32 bits of the scalar SEED will be used.

For more informations on the MT19937 RNG, see:

(1) **Matsumoto, M., and Nishimura, T., 1998:** Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. ACM Trans. on Modeling and Computer Simulation, Vol. 8, No. 1, January pp.3-30

6.17.29 `subroutine init_mt19937 ( seed )`

purpose

User interface subroutine for initializing the state of the MT19937 Random Number Generator (RNG) with an array of seeds, directly, without using the subroutine RANDOM_SEED_ and its interface.

Arguments

SEED (INPUT) integer(i4b), dimension(:) On entry, a vector of integers that will be used to initialize the MT19937 RNG. If size(SEED) is zero, a default scalar seed will be used instead.

Further Details

Only the first 32 bits of each element of the array SEED will be used.

For more informations on the MT19937 RNG, see:

(1) **Matsumoto, M., and Nishimura, T., 1998:** Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. ACM Trans. on Modeling and Computer Simulation, Vol. 8, No. 1, January pp.3-30

6.17.30 `subroutine init_memt19937 ( seed )`

purpose

User interface subroutine for initializing the state of the MEMT19937-II Random Number Generator (RNG) with a seed, directly, without using the subroutine RANDOM_SEED_ and its interface.

Arguments

SEED (INPUT) integer(i4b) On entry, a scalar integer that will be used to initialize the MEMT19937-II RNG.

Further Details

Only the first 32 bits of the scalar SEED will be used.

For more informations on the MEMT19937-II RNG, see:

(1) **Harase, S., 2014:** On the F2-linear relations of Mersenne Twister pseudorandom number generators. Mathematics and Computers in Simulation, Volume 100, Pages 103-113.

6.17.31 `subroutine init_memt19937 ( seed )`

purpose

User interface subroutine for initializing the state of the MEMT19937-II Random Number Generator (RNG) with an array of seeds, directly, without using the subroutine RANDOM_SEED_ and its interface.

Arguments

SEED (INPUT) integer(i4b), dimension(:) On entry, a vector of integers that will be used to initialize the MT19937 RNG. If size(SEED) is zero, a default scalar seed will be used instead.

Further Details

Only the first 32 bits of each element of the array SEED will be used.

For more informations on the MEMT19937-II RNG, see:

(1) **Harase, S., 2014:** On the F2-linear relations of Mersenne Twister pseudorandom number generators. Mathematics and Computers in Simulation, Volume 100, Pages 103-113.

6.17.32 `subroutine random_seed_ ( alg, size, put, get )`

purpose

User interface for seeding the random number routines in module RANDOM.

Syntax is like RANDOM_SEED intrinsic subroutine and a call to RANDOM_SEED_() without arguments initiates a non-repeatable reset of the seeds used by the random number subroutines in module RANDOM.

As for RANDOM_SEED intrinsic subroutine, no more than one argument may be specified in a call to RANDOM_SEED_ .

Arguments

ALG (INPUT, OPTIONAL) integer On entry, a scalar default integer to select the random number generator used in subsequent calls to subroutines RANDOM_NUMBER_ , RANDOM_INTEGER32_ , RANDOM_INTEGER31_ and functions RAND_NUMBER, RAND_INTEGER32 and RAND_INTEGER31. The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;

- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values the random number generator is not changed. The default value is the L'Ecuyer's LFSR113 random number generator (e.g. ALG=3).

SIZE (OUTPUT, OPTIONAL) integer On exit, the size of the seed array used by the random number generators.

PUT (INPUT, OPTIONAL) integer, dimension(:) On entry, a scalar default integer vector of size at least equal to the size of the seed array (as returned by a call to RANDOM_SEED_ with argument SIZE) that will be used to reset the seed for subsequent calls to subroutine RANDOM_NUMBER_.

GET (OUTPUT, OPTIONAL) integer, dimension(:) On exit, a scalar default integer vector, which is the current value of the seed array. Argument GET must be of size at least equal to the size of the seed array (as returned by a call to RANDOM_SEED_ with argument SIZE).

Further Details

This subroutine is not thread-safe and must not be called in parallel when OPENMP is used. On the other hand, the associated routines RAND_NUMBER, RANDOM_NUMBER_, RAND_INTEGER32, RANDOM_INTEGER32_, RAND_INTEGER31 and RANDOM_INTEGER31_ are thread-safe if used with OpenMP directives and their states will be consistent while called from multiple OpenMP threads.

The Marsaglia's KISS (Keep It Simple Stupid) random number generator combines:

1) The congruential generator x(n) = 69069 cdot x(n-1) + 1327217885 with a period of 2^32;
2) A 3-shift shift-register generator with a period of 2^32 - 1;
3) Two 16-bit multiply-with-carry generators with a period of 597273182964842497 > 2^59.

The overall period of this KISS random number generator exceeds 2^123. More details on this Marsaglia's KISS random number generator are available in the references (3) and (4). This generator is also the one used by the intrinsic subroutine RANDOM_NUMBER as implemented in the GNU gfortran compiler.

The "fast" version of the Marsaglia's KISS random number generator uses only add, shift, exclusive-or and 'and' operations to produces exactly the same 32-bit integer output, which C views as unsigned and Fortran views as signed integers. This version avoids multiplication and is probably faster. More details are available in the reference (5).

The LFSR113 random number generator is described in the reference (2). This random number generator has a period length of about 2^113.

The MT19937 Mersenne Twister random number generator is described in the reference (7). This random number generator has a period length of about 2^19937-1, and 623-dimensional equidistribution property is assured.

The MEMT19937-II Mersenne Twister random number generator is described in the reference (8). This random number generator has also a period length of about 2^19937-1, and a new set of parameters is introduced in the tempering phase of MT19937, which gives a maximally equidistributed Mersenne Twister random number generator.

Note that the size of the seed array varies according to the selected random number generator.

For all the random number generators described above, extended precision versions are also available to generate full precision random real numbers of kind STND (up to 63-bit precision), using the method described in the reference (6).

The FORTRAN versions of these random number generators as implemented here require that 32-bits integer type is available on your computer and that 32-bits integers are represented in base 2 with two's complement notation.

However, the LFSR113, MT19937 and MEMT19937-II Mersenne Twister random number generators will also work if only 64-bits integer type is available on your system, but in that case you must specify the CPP macro _RANDOM_NOINT32 at compilation. Note, however that the other random number generators will not work properly with 64-bits integer type so they cannot be used on such system.

The KISS random number generators also assumed that integer overflows do not cause crashes. These assumptions are checked before using these random number generators.

On the other hand, the LFSR113, MT19937 and MEMT19937-II random number generators do not use integer arithmetic and are free of such assumptions.

This subroutine is adapted from:

(1) **Hennecke, M., 1995:** A Fortran90 interface to random number generation. Computer Physics Communications, Volume 90, Number 1, 117-120

(2) **L'Ecuyer, P., 1999:** Tables of Maximally-Equidistributed Combined LFSR Generators. Mathematics of Computation, 68, 225, 261-269.

(3) **Marsaglia, G., 1999:** Random number generators for Fortran. Posted to the computer-programming-forum. See: http://computer-programming-forum.com/49-fortran/b89977aa62f72ee8.htm

(4) **Marsaglia, G., 2005:** Double precision RNGs. Posted to the electronic billboard sci.math.num-analysis. See: http://sci.tech-archive.net/Archive/sci.math.num-analysis/2005-11/msg00352.html

(5) **Marsaglia, G., 2007:** Fortran and C: United with a KISS. Posted to the Google comp.lang.forum . See: http://groups.google.co.uk/group/comp.lang.fortran/msg/6edb8ad6ec5421a5

(6) **Doornik, J.A, 2007:** Conversion of high-period random number to floating point. ACM Transactions on Modeling and Computer Simulation, Volume 17, Issue 1, Article No. 3.

(7) **Matsumoto, M., and Nishimura, T., 1998:** Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. ACM Trans. on Modeling and Computer Simulation, Vol. 8, No. 1, January pp.3-30

(8) **Harase, S., 2014:** On the F2-linear relations of Mersenne Twister pseudorandom number generators. Mathematics and Computers in Simulation, Volume 100, Pages 103-113.

6.17.33 `function normal_rand_number ()`

purpose

This function returns a Gaussian distributed random real number.

Arguments

None

Further Details

This function uses the Cumulative Density Function (CDF) inversion method to generate a Gaussian random real number. Starting with a random number produced by the STATPACK uniform random number generator that can produce random numbers with the uniform distribution over the continuous range (0, 1) (denoted U(0, 1)), the CDF method simply inverts the CDF of a standard Gaussian distribution to produce a standard Gaussian (e.g. a Gaussian distribution with mean zero and standard deviation one) random real number.

The inverse Gaussian CDF is approximated to high precision using rational approximations (polynomials with degree 2 and 3) by the subroutine PPND7 described in the reference (1). This method gives 7 decimal digits of accuracy in the range [10**(-316), 1-10**(-316)] if computations are done in double or higher precision.

For more details on Uniform and Gaussian random number generators or the approximation of the inverse Gaussian CDF used here, see:

(1) **Devroye, L., 1986:** Non-Uniform Random Variate Generation. Springer-Verlag, http://cg.scs.carleton.ca/~luc/rnbookindex.html, New York.

(2) **Thomas, D.B., Luk, W., Leong, P.H.W., and Villasenor, J.D., 2007:** Gaussian random number generators. ACM Comput. Surv. 39, 4, Article 11 (October 2007), 38 pages, DOI = 10.1145/1287620.1287622 (http://doi.acm.org/10.1145/1287620.1287622)

(3) **Wichura, M.J., 1988:** Algorithm AS 241: The percentage points of the normal distribution. Appl. Statis. 37, 3, 477-484.

6.17.34 `subroutine normal_random_number_ ( harvest )`

purpose

This subroutine returns a random real number HARVEST following the standard Gaussian distribution.

Arguments

HARVEST (OUTPUT) real(stnd) A Gaussian distributed random real number.

Further Details

This subroutine uses the Cumulative Density Function (CDF) inversion method to generate a Gaussian random real number. Starting with a random number produced by the STATPACK uniform random number generator that can produce random numbers with the uniform distribution over the continuous range (0, 1) (denoted U(0, 1)), the CDF method simply inverts the CDF of a standard Gaussian distribution to produce a standard Gaussian (e.g. a Gaussian distribution with mean zero and standard deviation one) random real number.

The inverse Gaussian CDF is approximated to high precision using rational approximations (polynomials with degree 2 and 3) by the subroutine PPND7 described in the reference (1). This method gives 7 decimal digits of accuracy in the range [10**(-316), 1-10**(-316)] if computations are done in double or higher precision.

For more details on Uniform and Gaussian random number generators or the approximation of the inverse Gaussian CDF used here, see:

(1) **Devroye, L., 1986:** Non-Uniform Random Variate Generation. Springer-Verlag, http://cg.scs.carleton.ca/~luc/rnbookindex.html, New York.

(2) **Thomas, D.B., Luk, W., Leong, P.H.W., and Villasenor, J.D., 2007:** Gaussian random number generators. ACM Comput. Surv. 39, 4, Article 11 (October 2007), 38 pages, DOI = 10.1145/1287620.1287622 (http://doi.acm.org/10.1145/1287620.1287622)

(3) **Wichura, M.J., 1988:** Algorithm AS 241: The percentage points of the normal distribution. Appl. Statis. 37, 3, 477-484.

6.17.35 `subroutine normal_random_number_ ( harvest )`

purpose

This subroutine returns a random vector HARVEST following the standard normal (Gaussian) distribution.

Arguments

HARVEST (OUTPUT) real(stnd), dimension(:) A Gaussian distributed random real vector.

Further Details

This subroutines uses the Cumulative Density Function (CDF) inversion method to generate Gaussian random numbers. Starting with random numbers produced by the STATPACK uniform random number generator that can produce random numbers with the uniform distribution over the continuous range (0, 1) (denoted U(0, 1)), the CDF method simply inverts the CDF of a standard Gaussian distribution to produce standard Gaussian (e.g. a Gaussian distribution with mean zero and standard deviation one) random numbers.

The inverse Gaussian CDF is approximated to high precision using rational approximations (polynomials with degree 2 and 3) by the subroutine PPND7 described in the reference (1). This method gives 7 decimal digits of accuracy in the range [10**(-316), 1-10**(-316)] if computations are done in double or higher precision.

For more details on Uniform and Gaussian random number generators or the approximation of the inverse Gaussian CDF used here, see:

(1) **Devroye, L., 1986:** Non-Uniform Random Variate Generation. Springer-Verlag, http://cg.scs.carleton.ca/~luc/rnbookindex.html, New York.

(2) **Thomas, D.B., Luk, W., Leong, P.H.W., and Villasenor, J.D., 2007:** Gaussian random number generators. ACM Comput. Surv. 39, 4, Article 11 (October 2007), 38 pages, DOI = 10.1145/1287620.1287622 (http://doi.acm.org/10.1145/1287620.1287622)

(3) **Wichura, M.J., 1988:** Algorithm AS 241: The percentage points of the normal distribution. Appl. Statis. 37, 3, 477-484.

6.17.36 `subroutine normal_random_number_ ( harvest )`

purpose

This subroutine returns a random matrix HARVEST following the standard normal (Gaussian) distribution.

Arguments

HARVEST (OUTPUT) real(stnd), dimension(:,:) A Gaussian distributed random real matrix.

Further Details

This subroutines uses the Cumulative Density Function (CDF) inversion method to generate Gaussian random real numbers. Starting with random numbers produced by the STATPACK uniform random number generator that can produce random numbers with the uniform distribution over the continuous range (0, 1) (denoted U(0, 1)), the CDF method simply inverts the CDF of a standard Gaussian distribution to produce standard Gaussian (e.g. a Gaussian distribution with mean zero and standard deviation one) random real numbers.

The inverse Gaussian CDF is approximated to high precision using rational approximations (polynomials with degree 2 and 3) by the subroutine PPND7 described in the reference (1). This method gives 7 decimal digits of accuracy in the range [10**(-316), 1-10**(-316)] if computations are done in double or higher precision.

For more details on Uniform and Gaussian random number generators or the approximation of the inverse Gaussian CDF used here, see :

(1) **Devroye, L., 1986:** Non-Uniform Random Variate Generation. Springer-Verlag, http://cg.scs.carleton.ca/~luc/rnbookindex.html, New York.

(2) **Thomas, D.B., Luk, W., Leong, P.H.W., and Villasenor, J.D., 2007:** Gaussian random number generators. ACM Comput. Surv. 39, 4, Article 11 (October 2007), 38 pages, DOI = 10.1145/1287620.1287622 (http://doi.acm.org/10.1145/1287620.1287622)

(3) **Wichura, M.J., 1988:** Algorithm AS 241: The percentage points of the normal distribution. Appl. Statis. 37, 3, 477-484.

6.17.37 `function normal_rand_number2 ()`

purpose

This function returns a Gaussian distributed real random number of kind extd.

Arguments

None

Further Details

This function uses the Cumulative Density Function (CDF) inversion method to generate a Gaussian random real number of kind extd. Starting with a random number produced by the STATPACK uniform random number generator that can produce random numbers with the uniform distribution over the continuous range (0, 1) (denoted U(0, 1)), the CDF method simply inverts the CDF of a standard Gaussian

distribution to produce a standard Gaussian (e.g. a Gaussian distribution with mean zero and standard deviation one) random real number of kind extd.

The inverse Gaussian CDF is approximated to high precision using rational approximations (polynomials with degree 7) by the subroutine PPND16 described in the reference (1). This method gives about 16 decimal digits of accuracy in the range [10**(-316), 1-10**(-316)] if computations are done in double or higher precision.

This function is more accurate than NORMAL_RAND_NUMBER function, but it is slower.

For more details on Uniform and Gaussian random number generators or the approximation of the inverse Gaussian CDF used here, see :

(1) **Devroye, L., 1986:** Non-Uniform Random Variate Generation. Springer-Verlag, http://cg.scs.carleton.ca/~luc/rnbookindex.html, New York.

(2) **Thomas, D.B., Luk, W., Leong, P.H.W., and Villasenor, J.D., 2007:** Gaussian random number generators. ACM Comput. Surv. 39, 4, Article 11 (October 2007), 38 pages, DOI = 10.1145/1287620.1287622 (http://doi.acm.org/10.1145/1287620.1287622)

(3) **Wichura, M.J., 1988:** Algorithm AS 241: The percentage points of the normal distribution. Appl. Statis. 37, 3, 477-484.

6.17.38 `subroutine normal_random_number2_ ( harvest )`

purpose

This subroutine returns a Gaussian distributed real random number of kind extd.

Arguments

HARVEST (OUTPUT) real(extd) A Gaussian distributed random real number of kind extd.

Further Details

This subroutine uses the Cumulative Density Function (CDF) inversion method to generate a Gaussian random ral number of kind extd. Starting with a random number produced by the STATPACK uniform random number generator that can produce random numbers with the uniform distribution over the continuous range (0, 1) (denoted U(0, 1)), the CDF method simply inverts the CDF of a standard Gaussian distribution to produce a standard Gaussian (e.g. a Gaussian distribution with mean zero and standard deviation one) random real number of kind extd.

The inverse Gaussian CDF is approximated to high precision using rational approximations (polynomials with degree 7) by the subroutine PPND16 described in the reference (1). This method gives about 16 decimal digits of accuracy in the range [10**(-316), 1-10**(-316)] if computations are done in double or higher precision.

For more details on Uniform and Gaussian random number generators or the approximation of the inverse Gaussian CDF used here, see :

(1) **Devroye, L., 1986:** Non-Uniform Random Variate Generation. Springer-Verlag, http://cg.scs.carleton.ca/~luc/rnbookindex.html, New York.

(2) **Thomas, D.B., Luk, W., Leong, P.H.W., and Villasenor, J.D., 2007:** Gaussian random number generators. ACM Comput. Surv. 39, 4, Article 11 (October 2007), 38 pages, DOI = 10.1145/1287620.1287622 (http://doi.acm.org/10.1145/1287620.1287622)

(3) **Wichura, M.J., 1988:** Algorithm AS 241: The percentage points of the normal distribution. Appl. Statis. 37, 3, 477-484.

6.17.39 subroutine normal_random_number2_ (harvest)

purpose

This subroutine returns a random real vector of kind extd following the standard normal (Gaussian) distribution.

Arguments

HARVEST (OUTPUT) real(extd), dimension(:) A Gaussian distributed random real vector of kind extd.

Further Details

This subroutines uses the Cumulative Density Function (CDF) inversion method to generate Gaussian random real numbers of kind extd. Starting with random numbers produced by the STATPACK uniform random number generator that can produce random numbers with the uniform distribution over the continuous range (0, 1) (denoted U(0, 1)), the CDF method simply inverts the CDF of a standard Gaussian distribution to produce standard Gaussian (e.g. a Gaussian distribution with mean zero and standard deviation one) random real numbers of kind extd.

The inverse Gaussian CDF is approximated to high precision using rational approximations (polynomials with degree 7) by the subroutine PPND16 described in the reference (1). This method gives about 16 decimal digits of accuracy in the range [10**(-316), 1-10**(-316)] if computations are done in double or higher precision.

For more details on Uniform and Gaussian random number generators or the approximation of the inverse Gaussian CDF used here, see :

(1) **Devroye, L., 1986:** Non-Uniform Random Variate Generation. Springer-Verlag, http://cg.scs.carleton.ca/~luc/rnbookindex.html, New York.

(2) **Thomas, D.B., Luk, W., Leong, P.H.W., and Villasenor, J.D., 2007:** Gaussian random number generators. ACM Comput. Surv. 39, 4, Article 11 (October 2007), 38 pages, DOI = 10.1145/1287620.1287622 (http://doi.acm.org/10.1145/1287620.1287622)

(3) **Wichura, M.J., 1988:** Algorithm AS 241: The percentage points of the normal distribution. Appl. Statis. 37, 3, 477-484.

6.17.40 subroutine normal_random_number2_ (harvest)

purpose

This subroutine returns a random real matrix of kind extd following the standard normal (Gaussian) distribution.

Arguments

HARVEST (OUTPUT) real(extd), dimension(:,:) A Gaussian distributed random real matrix of kind extd.

Further Details

This subroutines uses the Cumulative Density Function (CDF) inversion method to generate Gaussian random real numbers of kind extd. Starting with random numbers produced by the STATPACK uniform random number generator that can produce random numbers with the uniform distribution over the continuous range (0, 1) (denoted U(0, 1)), the CDF method simply inverts the CDF of a standard Gaussian distribution to produce standard Gaussian (e.g. a Gaussian distribution with mean zero and standard deviation one) random real numbers of kind extd.

The inverse Gaussian CDF is approximated to high precision using rational approximations (polynomials with degree 7) by the subroutine PPND16 described in the reference (1). This method gives about 16 decimal digits of accuracy in the range [10**(-316), 1-10**(-316)] if computations are done in double or higher precision.

For more details on Uniform and Gaussian random number generators or the approximation of the inverse Gaussian CDF used here, see :

(1) **Devroye, L., 1986:** Non-Uniform Random Variate Generation. Springer-Verlag, http://cg.scs.carleton.ca/~luc/rnbookindex.html, New York.

(2) **Thomas, D.B., Luk, W., Leong, P.H.W., and Villasenor, J.D., 2007:** Gaussian random number generators. ACM Comput. Surv. 39, 4, Article 11 (October 2007), 38 pages, DOI = 10.1145/1287620.1287622 (http://doi.acm.org/10.1145/1287620.1287622)

(3) **Wichura, M.J., 1988:** Algorithm AS 241: The percentage points of the normal distribution. Appl. Statis. 37, 3, 477-484.

6.17.41 `function normal_rand_number3 ()`

purpose

This function returns a Gaussian distributed random real number.

Arguments

None

Further Details

This function uses the classical Box-Muller method to generate a Gaussian random real number.

For more details on Uniform and Gaussian random number generators or the Box-Muller method used here, see :

(1) **Devroye, L., 1986:** Non-Uniform Random Variate Generation. Springer-Verlag, http://cg.scs.carleton.ca/~luc/rnbookindex.html, New York.

(2) **Thomas, D.B., Luk, W., Leong, P.H.W., and Villasenor, J.D., 2007:** Gaussian random number generators. ACM Comput. Surv. 39, 4, Article 11 (October 2007), 38 pages, DOI = 10.1145/1287620.1287622 (http://doi.acm.org/10.1145/1287620.1287622)

(3) **Brent, R.P., 1993:** Fast Normal Random Generators for vector processors. Report TR-CS-93-04, Computer Sciences Laboratory, Australian National University.

6.17.42 subroutine normal_random_number3_ (harvest)

purpose

This subroutine returns a random real number HARVEST following the standard Gaussian distribution.

Arguments

HARVEST (OUTPUT) real(stnd) A Gaussian distributed random real number.

Further Details

This subroutine uses the classical Box-Muller method to generate a Gaussian random real number.

For more details on Uniform and Gaussian random number generators or the Box-Muller method used here, see :

(1) **Devroye, L., 1986:** Non-Uniform Random Variate Generation. Springer-Verlag, http://cg.scs.carleton.ca/~luc/rnbookindex.html, New York.

(2) **Thomas, D.B., Luk, W., Leong, P.H.W., and Villasenor, J.D., 2007:** Gaussian random number generators. ACM Comput. Surv. 39, 4, Article 11 (October 2007), 38 pages, DOI = 10.1145/1287620.1287622 (http://doi.acm.org/10.1145/1287620.1287622)

(3) **Brent, R.P., 1993:** Fast Normal Random Generators for vector processors. Report TR-CS-93-04, Computer Sciences Laboratory, Australian National University.

6.17.43 subroutine normal_random_number3_ (harvest)

purpose

This subroutine returns a random real vector HARVEST following the standard normal (Gaussian) distribution.

Arguments

HARVEST (OUTPUT) real(stnd), dimension(:) A Gaussian distributed random real vector.

Further Details

This subroutine uses the classical Box-Muller method to generate Gaussian random real numbers. The computations are parallelized if OPENMP is used.

For more details on Uniform and Gaussian random number generators or the Box-Muller method used here, see :

(1) **Devroye, L., 1986:** Non-Uniform Random Variate Generation. Springer-Verlag, http://cg.scs.carleton.ca/~luc/rnbookindex.html, New York.

(2) **Thomas, D.B., Luk, W., Leong, P.H.W., and Villasenor, J.D., 2007:** Gaussian random number generators. ACM Comput. Surv. 39, 4, Article 11 (October 2007), 38 pages, DOI = 10.1145/1287620.1287622 (http://doi.acm.org/10.1145/1287620.1287622)

(3) **Brent, R.P., 1993:** Fast Normal Random Generators for vector processors. Report TR-CS-93-04, Computer Sciences Laboratory, Australian National University.

6.17.44 subroutine normal_random_number3_ (harvest)

purpose

This subroutine returns a random matrix HARVEST following the standard normal (Gaussian) distribution.

Arguments

HARVEST (OUTPUT) real(stnd), dimension(:,:) A Gaussian distributed random real matrix.

Further Details

This subroutine uses the classical Box-Muller method to generate Gaussian random real numbers. The computations are parallelized if OPENMP is used.

For more details on Uniform and Gaussian random number generators or the Box-Muller method used here, see :

(1) **Devroye, L., 1986:** Non-Uniform Random Variate Generation. Springer-Verlag, http://cg.scs.carleton.ca/~luc/rnbookindex.html, New York.

(2) **Thomas, D.B., Luk, W., Leong, P.H.W., and Villasenor, J.D., 2007:** Gaussian random number generators. ACM Comput. Surv. 39, 4, Article 11 (October 2007), 38 pages, DOI = 10.1145/1287620.1287622 (http://doi.acm.org/10.1145/1287620.1287622)

(3) **Brent, R.P., 1993:** Fast Normal Random Generators for vector processors. Report TR-CS-93-04, Computer Sciences Laboratory, Australian National University.

6.17.45 subroutine random_qr_cmp (mat, diagr, beta, fillr, initseed)

Purpose

RANDOM_QR_CMP generates the first k columns of a pseudo-random QR factorization of a hypothetical real n-by-n matrix MAT, whose elements follow independently a Laplace_Gauss(0;1) distribution (e.g. the standard normal distribution):

MAT = Q * R

where Q is a pseudo-random orthogonal matrix following the Haar distribution from the group of orthogonal matrices and R is upper triangular. The upper-diagonal elements of R follow a Laplace_Gauss(0;1) distribution (e.g. the standard normal distribution) and the squares of the diagonal elements of R, (R(i,i))**(2) follow a chi-squared distribution with n-i+1 degrees of freedom.

Arguments

MAT (OUTPUT) real(stnd), dimension(:,:) On exit, the elements above the diagonal of the array MAT contain the corresponding elements of R if FILLR is present and set to true; otherwise the upper-diagonal elements of MAT are not modified. The elements on and below the diagonal, with the arrays BETA and DIAGR, represent the first k columns (with k=size(MAT,2) and k<=n=size(MAT,1)) of a pseudo-random orthogonal matrix Q following the Haar distribution as a product of elementary reflectors and a diagonal matrix, see Further Details.

The shape of MAT must verify:

- size(MAT, 2) <= size(MAT, 1).

DIAGR (OUTPUT) real(stnd), dimension(:) On exit, the diagonal elements of the matrix R, see Further Details.

The size of DIAGR must verify:

- size(DIAGR) = size(MAT, 2) <= size(MAT, 1).

BETA (OUTPUT) real(stnd), dimension(:) On exit, the scalars factors of the elementary reflectors, see Further Details.

The size of BETA must verify:

- size(BETA) = size(DIAGR) = size(MAT, 2) <= size(MAT, 1).

FILLR (INPUT, OPTIONAL) logical(lgl) on entry, if FILLR is set to true, the super-diagonal elements of R are generated in MAT on exit. If FILLR is set to false, the super-diagonal elements of MAT are not modified or used.

The default is FILLR = false.

INITSEED (INPUT, OPTIONAL) logical(lgl) On entry, if INITSEED=true, a call to RANDOM_SEED_() without arguments is done in the subroutine, in order to initiate a non-repeatable reset of the seed used by the STATPACK random generator.

The default is INITSEED = false.

Further Details

RANDOM_QR_CMP uses the method described in the reference (1), based on Householder transformations, for generating the first k columns of a n-by-n pseudo-random orthogonal matrices Q distributed according to the Haar measure over the orthogonal group of order n, in a factored form.

The pseudo-random orthogonal matrix Q is represented as a product of n elementary reflectors and of a diagonal matrix

Q = H(1) * H(2) * . . . * H(n) * diag(sign(DIAGR))

Each H(i) has the form

H(i) = I + beta * (v * v') ,

where beta is a real scalar and v is a real n-element vector with v(1:i-1) = 0. v(i:n) is stored on exit in MAT(i:n,i) and beta in BETA(i).

diag(sign(DIAGR)) is the n-by-n diagonal matrix with diagonal elements equal to sign(one, DIAGR) (e.g., its ith diagonal element equals to one if DIAGR(i) is positive and -one otherwise).

It is possible to compute only the first k columns of Q and R by restricting the number of columns of MAT and the sizes of DIAGR and BETA on entry of the subroutine.

Finally, note that the computations are parallelized if OPENMP is used.

Q can be generated with the help of subroutine ORTHO_GEN_RANDOM_QR or can be applied to a vector or a matrix with the help of subroutine APPLY_Q_QR in module QR_Procedures.

For further details on the QR factorization and its use or pseudo-random orthogonal matrices distributed according to the Haar measure over the orthogonal group, see:

(1) **Stewart, G.W., 1980:** The efficient generation of random orthogonal matrices with an application to condition estimators. SIAM J. Numer. Anal., 17, 403-409

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.17.46 subroutine ortho_gen_random_qr (mat, diagr, beta)

Purpose

ORTHO_GEN_RANDOM_QR generates a n-by-n real pseudo-random orthogonal matrix following the Haar distribution, which is defined as the product of n elementary reflectors of order n and of a n-by-n diagonal matrix with diagonal elements equal to sign(one, DIAGR):

Q = H(1) * H(2) * . . . * H(n) * diag(sign(DIAGR))

as returned by RANDOM_QR_CMP.

Optionally, it is possible to generate only the first k columns of Q by restricting arguments MAT, BETA and DIAGR to the first k columns or elements of the corresponding arguments as returned by RANDOM_QR_CMP.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the i-th column must contain the vector which defines the elementary reflector H(i), for i = 1,2,. . . ,k, (with k<=n=size(MAT,1) and k=size(MAT,2)) as returned by RANDOM_QR_CMP in its array argument MAT. On exit, the first k columns of the pseudo-random n-by-n orthogonal matrix Q.

The shape of MAT must verify: size(MAT, 2) <= size(MAT, 1).

DIAGR (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the matrix R, as returned by RANDOM_QR_CMP in its argument DIAGR.

The size of DIAGR must verify: size(DIAGR) = size(MAT, 2).

BETA (INPUT) real(stnd), dimension(:) On entry, BETA(i) must contain the scalar factor of the elementary reflector H(i), as returned by RANDOM_QR_CMP in its argument BETA.

The size of BETA must verify: size(BETA) = size(DIAGR) = size(MAT, 2) .

Further Details

This subroutine used a blocked algorithm for agregating the Householder transformations (e.g. the elementary reflectors) stored in the lower triangle of MAT and generating the pseudo-random orthogonal matrix Q of the QR factorization returned by subroutine RANDOM_QR_CMP.

Furthermore, the computations are parallelized if OPENMP is used.

For further details on the QR factorization and its use or the blocked algorithm used here, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(4) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.17.47 subroutine gen_random_sym_mat (eigval, mat, eigvec, initseed, hous)

Purpose

GEN_RANDOM_SYM_MAT generates a pseudo-random n-by-n real symmetric matrix with prescribed eigenvalues.

Optionally, the corresponding eigenvectors of the generated pseudo-random n-by-n real symmetric matrix can be output if required.

Arguments

EIGVAL (INPUT) real(stnd), dimension(:) On entry, the prescribed eigenvalues of the pseudo-random n-by-n real symmetric matrix. IF size(EIGVAL)<n, the remaining eigenvalues are assumed to be zero.

The size of EIGVAL must verify:

- size(EIGVAL) <= size(MAT, 1) = size(MAT, 2) = n .

MAT (OUTPUT) real(stnd), dimension(:,:) On exit, the pseudo-random n-by-n real symmetric matrix.

The shape of MAT must verify:

- size(MAT, 2) = size(MAT, 1) = n .

EIGVEC (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, the pseudo-random eigenvectors corresponding to the eigenvalues prescribed in EIGVAL. The eigenvectors are returned columnwise.

The shape of EIGVEC must verify:

- size(EIGVEC, 1) = size(MAT, 1) = size(MAT, 2) = n ;
- size(EIGVEC, 2) = size(EIGVAL) .

INITSEED (INPUT, OPTIONAL) logical(lgl) On entry, if INITSEED=true, a call to RANDOM_SEED_() without arguments is done in the subroutine, in order to initiate a non-repeatable reset of the seed used by the STATPACK random generator.

The default is INITSEED = false.

HOUS (INPUT, OPTIONAL) logical(lgl) On entry, if:

- HOUS=true, the pseudo-random eigenvectors are generated as Householder transformation matrices ;

- HOUS=false, the pseudo-random eigenvectors are generated as pseudo-random orthogonal matrices following the Haar distribution from the group of orthogonal matrices.

The default is HOUS = true.

Further Details

Pseudo-random eigenvectors are generated as a pseudo-random orthogonal matrix following the Haar distribution from the group of orthogonal matrices and are computed with the help of subroutines RANDOM_QR_CMP and ORTHO_GEN_RANDOM_QR if the logical optional argument HOUS has the value false. If the logical optional argument HOUS has the value true, they are generated as Householder transformation matrices with the help of function GEN_RANDOM_ORTHO_MAT.

These computed pseudo-random eigenvectors and the corresponding prescribed eigenvalues are then used to generate a pseudo-random n-by-n symmetric matrix whose eigenvalues are the prescribed eigenvalues.

These computations are parallelized if OPENMP is used.

For further details, see:

For further details on Householder matrices and pseudo-random orthogonal matrices following the Haar distribution, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Stewart, G.W., 1980:** The efficient generation of random orthogonal matrices with an application to condition estimators. SIAM J. Numer. Anal., 17, 403-409

6.17.48 subroutine gen_random_mat (singval, mat, leftvec, rightvec, initseed, hous)

Purpose

GEN_RANDOM_MAT generates a pseudo-random m-by-n real matrix with prescribed singular values.

Optionally, the corresponding singular vectors of the generated pseudo-random m-by-n real matrix can be output if required.

Arguments

SINGVAL (INPUT) real(stnd), dimension(:) On entry , the prescribed singular values of the pseudo-random m-by-n real matrix.

The prescribed singular values must be greater or equal to zero. Furthermore, iF size(SINGVAL)<min(m,n), the remaining singular values are assumed to be zero.

The size of SINGVAL must verify:

- size(SINGVAL) <= min(size(MAT, 1), size(MAT, 2)) = min(m, n) .

MAT (OUTPUT) real(stnd), dimension(:,:) On exit, the pseudo-random m-by-n real matrix.

LEFTVEC (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, the pseudo-random left singular vectors corresponding to the singular values prescribed in SINGVAL. The left singular vectors are returned columnwise.

The shape of LEFTVEC must verify:

- size(LEFTVEC, 1) = size(MAT, 1) = m ;
- size(LEFTVEC, 2) = size(SINGVAL) .

RIGHTVEC (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, the pseudo-random right singular vectors corresponding to the singular values prescribed in SINGVAL. The right singular vectors are returned columnwise.

The shape of RIGHTVEC must verify:

- size(RIGHTVEC, 1) = size(MAT, 2) = n ;
- size(RIGHTVEC, 2) = size(SINGVAL) .

INITSEED (INPUT, OPTIONAL) logical(lgl) On entry, if INITSEED=true, a call to RANDOM_SEED_() without arguments is done in the subroutine, in order to initiate a non-repeatable reset of the seed used by the STATPACK random generator.

The default is INITSEED = false.

HOUS (INPUT, OPTIONAL) logical(lgl) On entry, if:

- HOUS=true, the pseudo-random singular vectors are generated as Householder transformation matrices ;
- HOUS=false, the pseudo-random singular vectors are generated as pseudo-random orthogonal matrices following the Haar distribution from the group of orthogonal matrices.

The default is HOUS = true.

Further Details

Pseudo-random singular vectors are generated as pseudo-random orthogonal matrices following the Haar distribution from the group of orthogonal matrices and are computed with the help of subroutines RANDOM_QR_CMP and ORTHO_GEN_RANDOM_QR if the logical optional argument HOUS has the value false. If the logical optional argument HOUS has the value true, they are generated as Householder transformation matrices with the help of function GEN_RANDOM_ORTHO_MAT.

These computed pseudo-random singular vectors and the corresponding prescribed singular values are then used to generate a pseudo-random m-by-n real matrix whose singular values are the prescribed singular values.

These computations are parallelized if OPENMP is used.

For further details on Householder matrices and pseudo-random orthogonal matrices following the Haar distribution, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Stewart, G.W., 1980:** The efficient generation of random orthogonal matrices with an application to condition estimators. SIAM J. Numer. Anal., 17, 403-409

6.17.49 function gen_random_ortho_mat (m, n, initseed)

Purpose

GEN_RANDOM_ORTHO_MAT generates a pseudo-random m-by-n real matrix with orthonormal columns.

Arguments

M (INPUT) integer(i4b) On entry, the number of rows of the pseudo-random m-by-n real orthonormal matrix.

N (INPUT) integer(i4b) On entry, the number of columns of the pseudo-random m-by-n real orthonormal matrix.

N must verify:

- N <= M .

INITSEED (INPUT, OPTIONAL) logical(lgl) On entry, if INITSEED=true, a call to RANDOM_SEED_() without arguments is done in the subroutine, in order to initiate a non-repeatable reset of the seed used by the STATPACK random generator.

The default is INITSEED = false.

Further Details

The pseudo-random m-by-n real matrix with orthonormal columns is generated from the first n columns of an m-by-m Householder matrix H (e.g. a Householder transformation), which is an orthogonal matrix defined as

H = I + tau * (v * v') ,

where tau is a real scalar and v is a real m-element vector. The scalar tau and vector v are computed to annilhate all the elements of a m-element pseudo-random uniform vector, excepted the first element.

These computations are parallelized if OPENMP is used.

For further details, on Householder matrices, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.17.50 subroutine partial_rqr_cmp (mat, diagr, beta, ip, krank, tol, tau, rng_alg, blk_size, nover)

Purpose

PARTIAL_RQR_CMP computes a randomized (partial or complete) QR factorization with column pivoting or orthogonal factorization of a m-by-n matrix MAT:

MAT * P = Q * R

, where P is a n-by-n permutation matrix, R is an upper triangular or trapezoidal (i.e., if n>m) m-by-n matrix and Q is a m-by-m orthogonal matrix.

At the user option, the randomized QR factorization can be only partial, e.g., the subroutine ends when the numbers of columns of Q is equal to a predefined value equals to kpartial = size(DIAGR) = size(BETA).

This leads implicitly to the following partition of Q:

[Q1 Q2]

where Q1 is a m-by-kpartial orthonormal matrix and Q2 is a m-by-(m-kpartial) orthonormal matrix orthogonal to Q1, and to the following corresponding partition of R:

[R11 R12]

[R21 R22]

where R11 is a kpartial-by-kpartial triangular matrix, R21 is zero by construction, R12 is a full kpartial-by-(n-kpartial) matrix and R22 is a full (m-kpartial)-by-(n-kpartial) matrix.

Then, if the optional scalar argument TOL is present and:

- is in]0,1[, the rank of R11 is determined by finding the submatrix of R11 which is defined as the largest leading submatrix whose estimated condition number, in the 1-norm, is less than 1/TOL.
- is equal to 0, the rank of R11, krank, is determined by finding the largest submatrix of R11 such that abs(R11[j,j])>0.

In both cases, the order of this submatrix, krank, is the effective rank of R11 (and MAT if krank is less than kpartial or if krank=kpartial=min(m,n)).

If TOL is absent or outside [0,1[, the rank of R11 is not checked and is assumed to be equal to kpartial.

If krank is less than kpartial, then MAT is not of full rank (i.e., certain columns of MAT(:m,:kpartial) are linear combinations of other columns of MAT(:m,:kpartial)) and krank is also an estimate of the rank of MAT.

This leads to a redefinition of the partition of Q = [Q1 Q2], where Q1 and Q2 are now m-by-krank and m-by-(m-krank) orthonormal matrices, and a corresponding redefinition of the associated partition of R, where R11 is now a krank-by-krank triangular matrix, R21 is again zero by construction, R12 is a full krank-by-(n-krank) matrix and R22 is a (m-krank)-by-(n-krank) matrix.

In a final step, if TAU is present, R22 is considered to be negligible and R12 is annihilated by orthogonal transformations from the right, arriving at the partial or complete orthogonal factorization:

MAT * P = Q * T * Z

, where P is a n-by-n permutation matrix, Q is a m-by-m orthogonal matrix, Z is a n-by-n orthogonal matrix and T is a m-by-n matrix and has the form:

[T11 T12]

[T21 T22]

Here T21 (=R21) and T12 are all zero, T22 (=R22) is considered to be negligible and T11 is a krank-by-krank upper triangular matrix.

On exit, P is stored compactly in the integer vector argument IP and if:

- TAU is absent, PARTIAL_RQR_CMP computes Q and submatrices R11, R12 and R22. Submatrices R11, R12 and R22 are stored in the array arguments MAT and DIAGR. Q is stored compactly in factored form in the array arguments MAT and BETA.
- TAU is present, PARTIAL_RQR_CMP computes Q, Z and submatrices T11 and T22 (=R22). Submatrices T11 and T22 are stored in the array arguments MAT and DIAGR. Q is stored compactly in factored form in the array arguments MAT and BETA. Z is stored compactly in factored form in the array arguments MAT and TAU.

See Further Details for more information on how the (partial or complete) QR or orthogonal decomposition is stored in MAT on exit.

PARTIAL_RQR_CMP performs the same task as subroutine PARTIAL_QR_CMP in module QR_Procedures, but is much faster on large matrices because of the use of a randomized and blocked "BLAS3" algorithm instead of a standard "BLAS2" algorithm. Note, however, that PARTIAL_RQR_CMP is an effective and efficient way for computing a low-rank approximation of MAT, but is less effective to find the rank of MAT because of the use of randomization. As an illustration, the diagonal elements of R11 are not necessarily of decreasing magnitude when computed by PARTIAL_RQR_CMP, while this property is enforced with PARTIAL_QR_CMP.

On the other hand, PARTIAL_QR_CMP can be used for both tasks, but is much slower than PARTIAL_RQR_CMP.

Note that PARTIAL_RQR_CMP performs also exactly the same task as subroutine PARTIAL_RQR_CMP2 in module Random. The main difference between the two routines is that PARTIAL_RQR_CMP2 recomputes the Gaussian and compression matrices at each iteration of the randomized (partial or complete) QR algorithm, while PARTIAL_RQR_CMP uses an efficient updating formulae to recompute the compression matrix at each iteration. In other words, PARTIAL_RQR_CMP is usually faster than PARTIAL_RQR_CMP2, but may be sligthly less robust. See references (3), (4), (5) and (6) for further information.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real m-by-n matrix to be decomposed.

On exit, MAT has been overwritten by details of its (partial) QR or orthogonal factorization.

See Further Details.

DIAGR (OUTPUT) real(stnd), dimension(:) On exit, the diagonal elements of the matrix R11 or T11.

See Further Details.

The size of DIAGR must verify:

- size(DIAGR) = kpartial <= min(size(MAT,1) , size(MAT,2)).

BETA (OUTPUT) real(stnd), dimension(:) On exit, the scalars factors of the elementary reflectors defining Q.

See Further Details.

The size of BETA must verify:

- size(BETA) = kpartial <= min(size(MAT,1) , size(MAT,2)).

IP (OUTPUT) integer(i4b), dimension(:) On exit, if IP(j)=k, then the j-th column of MAT*P was the k-th column of MAT.

See Further Details.

The size of IP must be equal to size(MAT, 2) = n.

KRANK (OUTPUT) integer(i4b) On exit, KRANK contains the effective rank of R11, i.e., krank, which is the order of this submatrix R11. This is the same as the order of the submatrix T11 in the complete orthogonal factorization of MAT and is also the rank of MAT if krank is less than kpartial = size(BETA).

If the computed pseudo-rank, krank, is less than kpartial = size(BETA), BETA(krank+1:kpartial) and, eventually, TAU(krank+1:kpartial) are set to zero and MAT(krank+1:m,krank+1:n) (e.g., R22=T22) is updated on exit.

In other words, the subroutine outputs a partial QR factorization of rank krank instead of rank kpartial.

In all cases, norm(MAT(krank+1:m,krank+1:n)) gives the error of the associated matrix approximation in the Frobenius norm, on exit.

TOL (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, if TOL is present then:

- if TOL is in]0,1[, the calculations to determine the condition number of R11 are performed. Then, TOL is used to determine the effective pseudo-rank of R11, which is defined as the order of the largest leading triangular submatrix in the partial QR factorization with column pivoting of MAT, whose estimated condition number in the 1-norm is less than 1/TOL. On exit, the reciprocal of the condition number is returned in TOL.
- if TOL=0 is specified, the calculations to determine the condition number of R11 are not performed and crude tests on R(j,j) are done to determine the numerical pseudo-rank of R11. On exit, TOL is not changed.

If TOL is not specified or is outside [0,1[, the calculations to determine the rank of R11 are not performed and this rank is assumed to be equal to kpartial.

TAU (OUTPUT, OPTIONAL) real(stnd), dimension(:) On entry, if TAU is present, a (partial) complete orthogonal factorization of MAT is computed. Otherwise, a simple QR factorization with column pivoting of MAT is computed.

On exit, the scalars factors of the elementary reflectors defining the orthogonal matrix Z in the (partial) orthogonal factorization of MAT.

See Further Details.

The size of TAU must verify:

- size(TAU) = kpartial <= min(size(MAT,1) , size(MAT,2)).

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian matrix in the randomized partial QR algorithm.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to PARTIAL_RQR_CMP.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

BLK_SIZE (INPUT, OPTIONAL) integer(i4b) On entry, the block size used in the randomized partial QR factorization.

BLK_SIZE must be greater or equal to one and less than min(m,n) and must be set to a much smaller value than min(m,n) usually, depending also on the architecture of the computer.

By default, BLK_SIZE is set to min(BLKSZ_QR, min(m,n)), where parameter BLKSZ_QR is the default block size for QR related algorithms specified in module Select_Parameters.

NOVER (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the randomized partial QR algorithm.

NOVER must be positive or null and verifies the relationship:

NOVER + BLK_SIZE <= size(MAT,1)

and is adjusted if necessary to verify this relationship in all cases.

See Further Details and the cited references for the meaning and usefulness of the oversampling size in the partial randomized QR algorithm.

By default, the oversampling size is set to 10.

Further Details

The matrix Q is represented as a product of elementary reflectors

Q = H(1) * H(2) * . . . * H(krank), where krank = size(BETA) <= min(m , n).

Each H(i) has the form

H(i) = I + beta * (v * v') ,

where beta is a real scalar and v is a real m-element vector with v(1:i-1) = 0. v(i:m) is stored on exit in MAT(i:m,i) and beta in BETA(i).

On exit of PARTIAL_RQR_CMP, the orthonormal matrix Q (or its first n columns) can be computed explicitly by a call to subroutine ORTHO_GEN_QR with arguments MAT and BETA.

The matrix P is represented in the array IP as follows: If IP(j) = i then the jth column of P is the ith canonical unit vector.

On exit, if the optional argument TAU is absent:

- The elements above the diagonal of the array MAT(:krank,:krank) contain the corresponding elements of the triangular matrix R11.
- The elements of the diagonal of R11 are stored in the array DIAGR.
- The submatrix R12 is stored in MAT(:krank,krank+1:n).
- The submatrix R22 is stored in MAT(krank+1:m,krank+1:n).
- krank is stored in the real argument KRANK.

If TAU is present, a partial or complete orthogonal factorization of MAT is computed. The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), which is used to introduce zeros into the kth row of R (e.g., in R12), is given in the form

[I 0]

[0 T(k)]

where

T(k) = I + tau * (u(k) * u(k)') and u(k)' = (1 0 z(k))

tau is a scalar, u(k) is a n-k+1 vector and z(k) is an (n-krank) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of R12.

The Z n-by-n orthogonal matrix is given by

Z = Z(1) * Z(2) * ... * Z(krank)

On exit, if the optional argument TAU is present:

- The scalar tau defining T(k) is returned in the kth element of TAU and the vector u(k) in the kth row of MAT, such that the elements of z(k) are in MAT(k,krank+1:n). The other elements of u(k) are not stored.
- The elements above the diagonal of the array section MAT(:krank,:krank) contain the corresponding elements of the triangular matrix T11. The elements of the diagonal of T11 are stored in the array DIAGR.
- The submatrix T22 (=R22) is stored in MAT(krank+1:m,krank+1:n).
- krank is stored in the real argument KRANK.

In both cases, norm(MAT(krank+1:m,krank+1:n)) gives the error of the associated matrix approximation in the Frobenius norm, on exit.

If it is possible that MAT may not be of full rank (i.e., certain columns of MAT are linear combinations of other columns), then the eventual linearly dependent columns in the (partial or complete) QR decomposition of MAT, which is sought, can be determined by using TOL=relative precision of the elements in MAT. If each element is correct to, say, 5 digits then TOL=0.00001 should be used. Also, it may be helpful to scale the columns of MAT so that all elements are about the same order of magnitude.

The computations are parallelized if OPENMP is used. Note also that PARTIAL_RQR_CMP uses a randomized "BLAS3" algorithm described in the references (3), (4), (5) and (6), which has about the same efficiency of a "BLAS3" QR algorithm without column pivoting and is thus highly efficient on large matrices.

For further details, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Duersch, J.A., and Gu, M., 2017:** Randomized QR with column pivoting. SIAM J. Sci. Comput., Volume 39, Issue 4, C263-C291.

(4) **Martinsson, P.G., Quintana-Orti, G., Heavner, N., and Van de Geijn, R., 2017:** Householder QR factorization with randomization for column pivoting (HQRRP). SIAM J. Sci. Comput., Volume 39, Issue 2, C96-C115.

(5) **Xiao, J., Gu, M., and Langou, J., 2017:** Fast parallel randomized QR with column pivoting algorithms for reliable low-rank matrix approximations. IEEE 24th International Conference on High Performance Computing (HiPC), IEEE, 2017, 233-242.

(6) **Duersch, J.A., and Gu, M., 2020:** Randomized projection for rank-revealing matrix factorizations and low-rank approximations. SIAM Review, Volume 62, Issue 3, 661-682.

6.17.51 subroutine partial_rqr_cmp2 (mat, diagr, beta, ip, krank, tol, tau, rng_alg, blk_size, nover)

Purpose

PARTIAL_RQR_CMP2 computes a randomized (partial or complete) QR factorization with column pivoting or orthogonal factorization of a m-by-n matrix MAT:

MAT * P = Q * R

, where P is a n-by-n permutation matrix, R is an upper triangular or trapezoidal (i.e., if n>m) m-by-n matrix and Q is a m-by-m orthogonal matrix.

At the user option, the randomized QR factorization can be only partial, e.g., the subroutine ends when the numbers of columns of Q is equal to a predefined value equals to kpartial = size(DIAGR) = size(BETA).

This leads implicitly to the following partition of Q:

[Q1 Q2]

where Q1 is a m-by-kpartial orthonormal matrix and Q2 is a m-by-(m-kpartial) orthonormal matrix orthogonal to Q1, and to the following corresponding partition of R:

[R11 R12]

[R21 R22]

where R11 is a kpartial-by-kpartial triangular matrix, R21 is zero by construction, R12 is a full kpartial-by-(n-kpartial) matrix and R22 is a full (m-kpartial)-by-(n-kpartial) matrix.

Then, if the optional scalar argument TOL is present and:

- is in]0,1[, the rank of R11 is determined by finding the submatrix of R11 which is defined as the largest leading submatrix whose estimated condition number, in the 1-norm, is less than 1/TOL.
- is equal to 0, the rank of R11, krank, is determined by finding the largest submatrix of R11 such that abs(R11[j,j])>0.

In both cases, the order of this submatrix, krank, is the effective rank of R11 (and MAT if krank is less than kpartial or if krank=kpartial=min(m,n)).

If TOL is absent or outside [0,1[, the rank of R11 is not checked and is assumed to be equal to kpartial.

If krank is less than kpartial, then MAT is not of full rank (i.e., certain columns of MAT(:m,:kpartial) are linear combinations of other columns of MAT(:m,:kpartial)) and krank is also an estimate of the rank of MAT.

This leads to a redefinition of the partition of Q = [Q1 Q2], where Q1 and Q2 are now m-by-krank and m-by-(m-krank) orthonormal matrices, and a corresponding redefinition of the associated partition of R, where R11 is now a krank-by-krank triangular matrix, R21 is again zero by construction, R12 is a full krank-by-(n-krank) matrix and R22 is a (m-krank)-by-(n-krank) matrix.

In a final step, if TAU is present, R22 is considered to be negligible and R12 is annihilated by orthogonal transformations from the right, arriving at the partial or complete orthogonal factorization:

MAT * P = Q * T * Z

, where P is a n-by-n permutation matrix, Q is a m-by-m orthogonal matrix, Z is a n-by-n orthogonal matrix and T is a m-by-n matrix and has the form:

[T11 T12]

[T21 T22]

Here T21 (=R21) and T12 are all zero, T22 (=R22) is considered to be negligible and T11 is a krank-by-krank upper triangular matrix.

On exit, P is stored compactly in the vector argument IP and if:

- TAU is absent, PARTIAL_RQR_CMP2 computes Q and submatrices R11, R12 and R22. Submatrices R11, R12 and R22 are stored in the array arguments MAT and DIAGR. Q is stored compactly in factored form in the array arguments MAT and BETA.
- TAU is present, PARTIAL_RQR_CMP2 computes Q, Z and submatrices T11 and T22 (=R22). Submatrices T11 and T22 are stored in the array arguments MAT and DIAGR. Q is stored compactly in factored form in the array arguments MAT and BETA. Z is stored compactly in factored form in the array arguments MAT and TAU.

See Further Details for more information on how the (partial or complete) QR or orthogonal decomposition is stored in MAT on exit.

PARTIAL_RQR_CMP2 performs the same task as subroutine PARTIAL_QR_CMP in module QR_Procedures, but is much faster on large matrices because of the use of a randomized and blocked "BLAS3" algorithm instead of a standard "BLAS2" algorithm. Note, however, that PARTIAL_RQR_CMP2 is an effective and efficient way for computing a low-rank approximation of MAT, but is less effective to find the rank of MAT because of the use of randomization. As an illustration, the diagonal elements of R11 are not necessarily of decreasing magnitude when computed by PARTIAL_RQR_CMP2, while this property is enforced with PARTIAL_QR_CMP.

On the other hand, PARTIAL_QR_CMP can be used for both tasks, but is much slower than PARTIAL_RQR_CMP2.

Note that PARTIAL_RQR_CMP2 performs also exactly the same task as subroutine PARTIAL_RQR_CMP in module Random. The main difference between the two routines is that PARTIAL_RQR_CMP2 recomputes the Gaussian and compression matrices at each iteration of the randomized (partial or complete) QR algorithm, while PARTIAL_RQR_CMP uses an efficient updating formulae to recompute the compression matrix at each iteration. In other words, PARTIAL_RQR_CMP is usually faster than PARTIAL_RQR_CMP2, but may be sligthly less robust. See the references (3), (4), (5) and (6) for further information.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real m-by-n matrix to be decomposed.

On exit, MAT has been overwritten by details of its (partial) QR or orthogonal factorization.

See Further Details.

DIAGR (OUTPUT) real(stnd), dimension(:) On exit, the diagonal elements of the matrix R11 or T11.

See Further Details.

The size of DIAGR must verify:

- size(DIAGR) = kpartial <= min(size(MAT,1) , size(MAT,2)).

BETA (OUTPUT) real(stnd), dimension(:) On exit, the scalars factors of the elementary reflectors defining Q.

See Further Details.

The size of BETA must verify:

- size(BETA) = kpartial <= min(size(MAT,1) , size(MAT,2)).

IP (OUTPUT) integer(i4b), dimension(:) On exit, if IP(j)=k, then the j-th column of MAT*P was the k-th column of MAT.

See Further Details.

The size of IP must be equal to size(MAT, 2) = n.

KRANK (OUTPUT) integer(i4b) On exit, KRANK contains the effective rank of R11, i.e., krank, which is the order of this submatrix R11. This is the same as the order of the submatrix T11 in the complete orthogonal factorization of MAT and is also the rank of MAT if krank is less than kpartial = size(BETA).

If the computed pseudo-rank, krank, is less than kpartial = size(BETA), BETA(krank+1:kpartial) and, eventually, TAU(krank+1:kpartial) are set to zero and MAT(krank+1:m,krank+1:n) (e.g., R22=T22) is updated on exit. In other words, the subroutine outputs a partial QR factorization of rank krank instead of rank kpartial.

In all cases, norm(MAT(krank+1:m,krank+1:n)) gives the error of the associated matrix approximation in the Frobenius norm, on exit.

TOL (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, if TOL is present then:

- if TOL is in]0,1[, the calculations to determine the condition number of R11 are performed. Then, TOL is used to determine the effective pseudo-rank of R11, which is defined as the order of the largest leading triangular submatrix in the partial QR factorization with column pivoting of MAT, whose estimated condition number in the 1-norm is less than 1/TOL. On exit, the reciprocal of the condition number is returned in TOL.
- if TOL=0 is specified, the calculations to determine the condition number of R11 are not performed and crude tests on R(j,j) are done to determine the numerical pseudo-rank of R11. On exit, TOL is not changed.

If TOL is not specified or is outside [0,1[, the calculations to determine the rank of R11 are not performed and this rank is assumed to be equal to kpartial.

TAU (OUTPUT, OPTIONAL) real(stnd), dimension(:) On entry, if TAU is present, a (partial) orthogonal factorization of MAT is computed. Otherwise, a simple QR factorization with column pivoting of MAT is computed.

On exit, the scalars factors of the elementary reflectors defining the orthogonal matrix Z in the (partial) orthogonal factorization of MAT.

See Further Details.

The size of TAU must verify:

- size(TAU) = kpartial <= min(size(MAT,1) , size(MAT,2)).

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian matrix in the randomized partial QR algorithm.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;

- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to PARTIAL_RQR_CMP2.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

BLK_SIZE (INPUT, OPTIONAL) integer(i4b) On entry, the block size used in the randomized partial QR factorization.

BLK_SIZE must be greater or equal to one and less than min(m,n) and must be set to a much smaller value than min(m,n) usually, depending also on the architecture of the computer.

By default, BLK_SIZE is set to min(BLKSZ_QR, min(m,n)), where parameter BLKSZ_QR is the default block size for QR related algorithms specified in module Select_Parameters.

NOVER (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the randomized partial QR algorithm.

NOVER must be positive or null and verifies the relationship:

NOVER + BLK_SIZE <= size(MAT,1)

and is adjusted if necessary to verify this relationship in all cases.

See Further Details and the cited references for the meaning and usefulness of the oversampling size in the partial randomized QR algorithm.

By default, the oversampling size is set to 10.

Further Details

The matrix Q is represented as a product of elementary reflectors

Q = H(1) * H(2) * . . . * H(krank), where krank = size(BETA) <= min(m , n).

Each H(i) has the form

H(i) = I + beta * (v * v') ,

where beta is a real scalar and v is a real m-element vector with v(1:i-1) = 0. v(i:m) is stored on exit in MAT(i:m,i) and beta in BETA(i).

On exit of PARTIAL_RQR_CMP2, the orthonormal matrix Q (or its first n columns) can be computed explicitly by a call to subroutine ORTHO_GEN_QR with arguments MAT and BETA.

The matrix P is represented in the array IP as follows: If IP(j) = i then the jth column of P is the ith canonical unit vector.

On exit, if the optional argument TAU is absent:

- The elements above the diagonal of the array MAT(:krank,:krank) contain the corresponding elements of the triangular matrix R11.
- The elements of the diagonal of R11 are stored in the array DIAGR.
- The submatrix R12 is stored in MAT(:krank,krank+1:n).

- The submatrix R22 is stored in MAT(krank+1:m,krank+1:n).
- krank is stored in the real argument KRANK.

If TAU is present, a partial or complete orthogonal factorization of MAT is computed. The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), which is used to introduce zeros into the kth row of R (e.g., in R12), is given in the form

[I 0]

[0 T(k)]

where

T(k) = I + tau * (u(k) * u(k)') and u(k)' = (1 0 z(k))

tau is a scalar, u(k) is a n-k+1 vector and z(k) is an (n-krank) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of R12.

The Z n-by-n orthogonal matrix is given by

Z = Z(1) * Z(2) * ... * Z(krank)

On exit, if the optional argument TAU is present:

- The scalar tau defining T(k) is returned in the kth element of TAU and the vector u(k) in the kth row of MAT, such that the elements of z(k) are in MAT(k,krank+1:n). The other elements of u(k) are not stored.
- The elements above the diagonal of the array section MAT(:krank,:krank) contain the corresponding elements of the triangular matrix T11. The elements of the diagonal of T11 are stored in the array DIAGR.
- The submatrix T22 (=R22) is stored in MAT(krank+1:m,krank+1:n).
- krank is stored in the real argument KRANK.

In both cases, norm(MAT(krank+1:m,krank+1:n)) gives the error of the associated matrix approximation in the Frobenius norm, on exit.

If it is possible that MAT may not be of full rank (i.e., certain columns of MAT are linear combinations of other columns), then the eventual linearly dependent columns in the partial QR decomposition of MAT, which is sought, can be determined by using TOL=relative precision of the elements in MAT. If each element is correct to, say, 5 digits then TOL=0.00001 should be used. Also, it may be helpful to scale the columns of MAT so that all elements are about the same order of magnitude.

The computations are parallelized if OPENMP is used. Note also that PARTIAL_RQR_CMP2 uses a randomized "BLAS3" algorithm described in the references (3), (4), (5) and (6), which has about the same efficiency of a "BLAS3" QR algorithm without column pivoting and is thus highly efficient on large matrices.

For further details, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Duersch, J.A., and Gu, M., 2017:** Randomized QR with column pivoting. SIAM J. Sci. Comput., Volume 39, Issue 4, C263-C291.

(4) **Martinsson, P.G., Quintana-Orti, G., Heavner, N., and Van de Geijn, R., 2017:** Householder QR factorization with randomization for column pivoting (HQRRP). SIAM J. Sci. Comput., Volume 39, Issue 2, C96-C115.

(5) **Xiao, J., Gu, M., and Langou, J., 2017:** Fast parallel randomized QR with column pivoting algorithms for reliable low-rank matrix approximations. IEEE 24th International Conference on High Performance Computing (HiPC), IEEE, 2017, 233-242.

(6) **Duersch, J.A., and Gu, M., 2020:** Randomized projection for rank-revealing matrix factorizations and low-rank approximations. SIAM Review, Volume 62, Issue 3, 661-682.

6.17.52 subroutine partial_rtqr_cmp (mat, diagr, beta, ip, krank, tol, tau, rng_alg, niter, nover)

Purpose

PARTIAL_RTQR_CMP computes a randomized partial and truncated QR factorization with column pivoting, or an orthogonal factorization, of a m-by-n matrix MAT:

MAT * P = Q * R

, where P is a n-by-n permutation matrix, R is an upper triangular or trapezoidal kpartial-by-n matrix and Q is a m-by-kpartial matrix with orthogonal columns.

The randomized QR factorization is only partial, e.g., the subroutine ends when the numbers of columns of Q is equal to a predefined value equals to kpartial = size(DIAGR) = size(BETA) <= min(m,n).

This leads implicitly to the following partition of R:

[R11 R12]

where R11 is a kpartial-by-kpartial triangular matrix and R12 is a full kpartial-by-(n-kpartial) matrix.

Then, if the optional scalar argument TOL is present and:

- is in]0,1[, the rank of R11 is determined by finding the submatrix of R11 which is defined as the largest leading submatrix whose estimated condition number, in the 1-norm, is less than 1/TOL.
- is equal to 0, the rank of R11, krank, is determined by finding the largest submatrix of R11 such that abs(R11[j,j])>0.

In both cases, the order of this submatrix, krank, is the effective rank of R11 (and MAT if krank is less than kpartial or if krank=kpartial=min(m,n)).

If TOL is absent or outside [0,1[, the rank of R11 is not checked and is assumed to be equal to kpartial.

If krank is less than kpartial, then MAT is not of full rank (i.e., certain columns of MAT(:m,:kpartial) are linear combinations of other columns of MAT(:m,:kpartial)) and krank is also an estimate of the rank of MAT.

This leads to a redefinition of the partition of Q = [Q1 Q2], where Q1 and Q2 are now m-by-krank and m-by-(kpartial-krank) orthonormal matrices, and a corresponding redefinition of the associated partition of R, where R11 is now a krank-by-krank triangular matrix and R12 is a full krank-by-(n-krank) matrix.

In a final step, if TAU is present, R12 is annihilated by orthogonal transformations from the right, arriving at the partial orthogonal factorization:

MAT * P = Q * T * Z

, where P is a n-by-n permutation matrix, Q is a m-by-krank orthogonal matrix, Z is a n-by-n orthogonal matrix and T is a krank-by-n matrix and has the form:

[T11 T12]

Here T12 is all zero and T11 is a krank-by-krank upper triangular matrix.

On exit, P is stored compactly in the vector argument IP and if:

- TAU is absent, PARTIAL_RTQR_CMP computes Q and submatrices R11 and R12. Submatrices R11, and R12 are stored in the array arguments MAT and DIAGR. Q is stored compactly in factored form in the array arguments MAT and BETA.
- TAU is present, PARTIAL_RTQR_CMP computes Q, Z and submatrix T11. Submatrix T11 is stored in the array arguments MAT and DIAGR. Q is stored compactly in factored form in the array arguments MAT and BETA. Z is stored compactly in factored form in the array arguments MAT and TAU.

See Further Details for more information on how the partial QR or orthogonal decomposition is stored in MAT on exit.

PARTIAL_RTQR_CMP performs the same task as subroutines PARTIAL_RQR_CMP and PARTIAL_RQR_CMP2 subroutines in module Random, but is significantly faster when krank is relatively small and MAT is a very large matrix. This is due to the fact that PARTIAL_RTQR_CMP computes Q (in factored form), R11 and R12, but not R22 (where R22 is the bottom left (m-krank)-by-(n-krank) submatrix in the QR factorization of MAT) as PARTIAL_RQR_CMP and PARTIAL_RQR_CMP2 subroutines. Furthermore, only an estimate of R12 is computed by PARTIAL_RTQR_CMP, using a randomization algorithm described in the reference (7), while the computation of R12 is exact in PARTIAL_RQR_CMP and PARTIAL_RQR_CMP2 subroutines.

In other words, PARTIAL_RTQR_CMP avoids trailing updates of MAT (e.g., R22) during the randomized QR factorization, which reduces significantly the CPU time compared to PARTIAL_RQR_CMP and PARTIAL_RQR_CMP2 subroutines, if krank is small and MAT is a very large matrix, as these subroutines perform these trailing updates.

Note that PARTIAL_RTQR_CMP also does not recompute or update the compression matrix at each iteration as PARTIAL_RQR_CMP and PARTIAL_RQR_CMP2 subroutines. See references (3), (4), (5), (6) and (7) for further information.

In summary, PARTIAL_RTQR_CMP subroutine is usually faster than PARTIAL_RQR_CMP and PARTIAL_RQR_CMP2 subroutines but is less accurate, especially for matrices with a slow decay of their singular values. See reference (7) for details.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real m-by-n matrix to be decomposed.

On exit, MAT has been overwritten by details of its partial and truncated QR or orthogonal factorization.

See Further Details.

DIAGR (OUTPUT) real(stnd), dimension(:) On exit, the diagonal elements of the matrix R11 or T11.

See Further Details.

The size of DIAGR must verify:

- size(DIAGR) = kpartial <= min(size(MAT,1) , size(MAT,2)).

BETA (OUTPUT) real(stnd), dimension(:) On exit, the scalars factors of the elementary reflectors defining Q.

See Further Details.

The size of BETA must verify:

- size(BETA) = kpartial <= min(size(MAT,1) , size(MAT,2)).

IP (OUTPUT) integer(i4b), dimension(:) On exit, if IP(j)=k, then the j-th column of MAT*P was the k-th column of MAT.

See Further Details.

The size of IP must be equal to size(MAT, 2) = n.

KRANK (OUTPUT) integer(i4b) On exit, KRANK contains the effective rank of R11, i.e., krank, which is the order of this submatrix R11. This is the same as the order of the submatrix T11 in the complete orthogonal factorization of MAT and is also the rank of MAT if krank is less than kpartial = size(BETA).

TOL (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, if TOL is present then:

- if TOL is in]0,1[, the calculations to determine the condition number of R11 are performed. Then, TOL is used to determine the effective pseudo-rank of R11, which is defined as the order of the largest leading triangular submatrix in the partial QR factorization with column pivoting of MAT, whose estimated condition number in the 1-norm is less than 1/TOL. On exit, the reciprocal of the condition number is returned in TOL.
- if TOL=0 is specified, the calculations to determine the condition number of R11 are not performed and crude tests on R(j,j) are done to determine the numerical pseudo-rank of R11. On exit, TOL is not changed.

If TOL is not specified or is outside [0,1[, the calculations to determine the rank of R11 are not performed and this rank is assumed to be equal to kpartial.

TAU (OUTPUT, OPTIONAL) real(stnd), dimension(:) On entry, if TAU is present, a (partial) complete orthogonal factorization of MAT is computed. Otherwise, a simple QR factorization with column pivoting of MAT is computed.

On exit, the scalars factors of the elementary reflectors defining the orthogonal matrix Z in the partial orthogonal factorization of MAT.

See Further Details.

The size of TAU must verify:

- size(TAU) = kpartial <= min(size(MAT,1) , size(MAT,2)).

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian matrix in the randomized partial QR algorithm.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to PARTIAL_RTQR_CMP.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

NITER (INPUT, OPTIONAL) integer(i4b) The number of randomized subspace iterations performed in the subroutine for improving the accuracy of the compression matrix before computing its partial QR factorization with column pivoting.

NITER must be positive or null.

By default, 0 randomized subspace iterations are performed.

NOVER (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the randomized partial QR algorithm.

NOVER must be positive or null and verifies the relationship:

NOVER + kpartial <= size(MAT,1)

and is adjusted if necessary to verify this relationship in all cases.

See Further Details and the cited references for the meaning and usefulness of the oversampling size in the randomized partial and truncated QR algorithm.

By default, the oversampling size is set to max(kpartial/2_i4b, 10).

Further Details

The matrix Q is represented as a product of elementary reflectors

Q = H(1) * H(2) * . . . * H(krank), where krank <= min(m , n).

Each H(i) has the form

H(i) = I + beta * (v * v') ,

where beta is a real scalar and v is a real m-element vector with v(1:i-1) = 0. v(i:m) is stored on exit in MAT(i:m,i) and beta in BETA(i).

On exit of PARTIAL_RTQR_CMP, the orthonormal matrix Q (or its first krank columns) can be computed explicitly by a call to subroutine ORTHO_GEN_QR with arguments MAT(:m,:krank) and BETA(:krank).

The matrix P is represented in the array IP as follows: If IP(j) = i then the jth column of P is the ith canonical unit vector.

On exit, if the optional argument TAU is absent:

- The elements above the diagonal of the array MAT(:krank,:krank) contain the corresponding elements of the triangular matrix R11.
- The elements of the diagonal of R11 are stored in the array DIAGR.
- The approximation of the submatrix R12 is stored in MAT(:krank,krank+1:n).
- krank is stored in the real argument KRANK.

If TAU is present, a partial complete orthogonal factorization of MAT is computed. The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), which is used to introduce zeros into the kth row of R (e.g., in R12), is given in the form

[I 0]

[0 T(k)]

where

T(k) = I + tau * (u(k) * u(k)') and u(k)' = (1 0 z(k))

tau is a scalar, u(k) is a n-k+1 vector and z(k) is an (n-krank) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of R12.

The Z n-by-n orthogonal matrix is given by

Z = Z(1) * Z(2) * ... * Z(krank)

On exit, if the optional argument TAU is present:

- The scalar tau defining T(k) is returned in the kth element of TAU and the vector u(k) in the kth row of MAT, such that the elements of z(k) are in MAT(k,krank+1:n). The other elements of u(k) are not stored.
- The elements above the diagonal of the array section MAT(:krank,:krank) contain the corresponding elements of the triangular matrix T11. The elements of the diagonal of T11 are stored in the array DIAGR.
- krank is stored in the real argument KRANK.

If it is possible that MAT may not be of full rank (i.e., certain columns of MAT are linear combinations of other columns), then the eventual linearly dependent columns in the partial QR decomposition of MAT, which is sought, can be determined by using TOL=relative precision of the elements in MAT. If each element is correct to, say, 5 digits then TOL=0.00001 should be used. Also, it may be helpful to scale the columns of MAT so that all elements are about the same order of magnitude.

The computations are parallelized if OPENMP is used. Note also that PARTIAL_RTQR_CMP uses a randomized "BLAS3" algorithm described in the reference (7), which has about the same efficiency of a "BLAS3" QR algorithm without column pivoting and is thus highly efficient on large matrices.

For further details, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Duersch, J.A., and Gu, M., 2017:** Randomized QR with column pivoting. SIAM J. Sci. Comput., Volume 39, Issue 4, C263-C291.

(4) **Martinsson, P.G., Quintana-Orti, G., Heavner, N., and Van de Geijn, R., 2017:** Householder QR factorization with randomization for column pivoting (HQRRP). SIAM J. Sci. Comput., Volume 39, Issue 2, C96-C115.

(5) **Xiao, J., Gu, M., and Langou, J., 2017:** Fast parallel randomized QR with column pivoting algorithms for reliable low-rank matrix approximations. IEEE 24th International Conference on High Performance Computing (HiPC), IEEE, 2017, 233-242.

(6) **Duersch, J.A., and Gu, M., 2020:** Randomized projection for rank-revealing matrix factorizations and low-rank approximations. SIAM Review, Volume 62, Issue 3, 661-682.

(7) **Mary, T., Yamazaki, I., Kurzak, J., Luszczek, P., Tomov, S., and Dongarra, J., 2015:** Performance of Random Sampling for Computing Low-rank Approximations of a Dense Matrix on GPUs. International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15).

6.17.53 subroutine partial_rqr_cmp_fixed_precision (mat, relerr, diagr, beta, ip, krank, tau, rng_alg, blk_size, nover)

Purpose

PARTIAL_RQR_CMP_FIXED_PRECISION computes a randomized partial QR factorization with column pivoting or orthogonal factorization of a m-by-n matrix MAT:

MAT * P = Q * R

, where P is a n-by-n permutation matrix, R is a krank-by-n (upper trapezoidal) matrix and Q is a m-by-krank matrix with orthonormal columns. This leads to the following matrix approximation of MAT of rank krank:

MAT = Q * (R * P')

krank is the target rank of the matrix approximation, which is sought, and this partial factorization must have an approximation error which fulfills:

|| MAT - Q * (R * P') ||_F <= ||MAT||_F * relerr

|| ||_F is the Frobenius norm and relerr is a prescribed accuracy tolerance for the relative error of the computed matrix approximation, specified in the input argument RELERR.

PARTIAL_RQR_CMP_FIXED_PRECISION searches incrementally the best (e.g., of smallest rank) Q * (R * P') approximation, which fulfills the prescribed accuracy tolerance for the relative error. More precisely, the rank of the matrix approximation is increased progressively of BLK_SIZE by BLK_SIZE until the prescribed accuracy tolerance is satisfied.

In other words, the rank, krank, of the matrix approximation is not known in advance and is determined in the subroutine. krank is stored in the argument KRANK and the relative error of the computed matrix approximation is output in argument RELERR on exit.

The computed matrix approximation leads implicitly to the following partition of R:

[R1 R2]

where R1 is a krank-by-krank triangular matrix and R2 is a full krank-by-(n-krank) matrix.

In a final step, if TAU is present, R2 is annihilated by orthogonal transformations from the right, arriving at the partial orthogonal factorization:

MAT * P = Q * T1 * Z

, where P is a n-by-n permutation matrix, Q is a m-by-krank matrix with orthonormal columns, Z is a krank-by-n matrix with orthonormal rows and T1 is a krank-by-krank upper triangular matrix.

Note, however, that this final step does not change the matrix approximation and its relative error, only the output format of this matrix approximation, which is now composed of four factors instead of three.

On exit, P is stored compactly in the vector argument IP, krank is stored in the scalar argument KRANK and if:

- TAU is absent, PARTIAL_RQR_CMP_FIXED_PRECISION computes Q and submatrices R1 and R2. Submatrices R1 and R2 are stored in the array arguments MAT and DIAGR(:KRANK). Q is stored compactly in factored form in the array arguments MAT and BETA(:KRANK).
- TAU is present, PARTIAL_RQR_CMP_FIXED_PRECISION computes Q, Z and submatrice T1. Submatrice T1 is stored in the array arguments MAT and DIAGR. Q is stored compactly in factored form in the array arguments MAT and BETA(:KRANK). Z is stored compactly in factored form in the array arguments MAT and TAU(:KRANK).

In all cases, the relative error of the computed matrix approximation is output in argument RELERR.

See Further Details for more information.

PARTIAL_RQR_CMP_FIXED_PRECISION performs the same task as subroutine PARTIAL_QR_CMP_FIXED_PRECISION in module QR_Procedures, but is much faster on large matrices because of the use of a randomized and blocked "BLAS3" algorithm instead of a standard "BLAS2" algorithm. Another difference is that, in PARTIAL_RQR_CMP_FIXED_PRECISION, the rank of the matrix approximation is increased progressively of BLK_SIZE by BLK_SIZE until the prescribed tolerance for the relative error is satisfied while in PARTIAL_QR_CMP_FIXED_PRECISION subroutine, the rank of the matrix approximation is increased one by one until the prescribed tolerance for the relative error is satisfied. In other words, the rank of the matrix approximation found by PARTIAL_RQR_CMP_FIXED_PRECISION is always larger than the one found by PARTIAL_QR_CMP_FIXED_PRECISION and is a multiple of BLK_SIZE.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real m-by-n matrix to be decomposed.

On exit, MAT has been overwritten by details of its partial QR or orthogonal factorization.

See Further Details.

RELERR (INPUT/OUTPUT) real(stnd) On entry, the requested accuracy tolerance for the relative error of the computed partial matrix approximation.

The preset value for RELERR must be greater than 4*epsilon(RELERR) and less than one.

On exit, RELERR contains the relative error of the computed partial matrix approximation:

- RELERR = || MAT - Q * (R * P') ||_F / ||MAT||_F

DIAGR (OUTPUT) real(stnd), dimension(:) On exit, the diagonal elements of the matrix R1 or T1 are stored in the array section DIAGR(:KRANK). Other elements of DIAGR are set to zero on exit.

See Further Details.

The size of DIAGR must verify:

- size(DIAGR) = min(size(MAT,1) , size(MAT,2)).

BETA (OUTPUT) real(stnd), dimension(:) On exit, the scalars factors of the elementary reflectors defining Q are stored in the array section BETA(:KRANK). Other elements of BETA are set to zero on exit.

See Further Details.

The size of BETA must verify:

- size(BETA) = min(size(MAT,1) , size(MAT,2)).

IP (OUTPUT) integer(i4b), dimension(:) On exit, if IP(j)=k, then the j-th column of MAT*P was the k-th column of MAT.

See Further Details.

The size of IP must be equal to size(MAT, 2) = n.

KRANK (OUTPUT) integer(i4b) On exit, KRANK contains the rank of R1, i.e., krank, which is the order of this submatrix R1. This is the same as the order of the submatrix T1 in the "partial" complete orthogonal factorization of MAT and is also the rank of the computed matrix approximation.

In all cases, norm(MAT(KRANK+1:m,KRANK+1:n)) gives the error of the associated matrix approximation in the Frobenius norm, on exit.

TAU (OUTPUT, OPTIONAL) real(stnd), dimension(:) On entry, if TAU is present, a partial complete orthogonal factorization of MAT is computed. Otherwise, a simple QR factorization with column pivoting of MAT is computed.

On exit, the scalars factors of the elementary reflectors defining the orthogonal matrix Z in the partial complete orthogonal factorization of MAT are stored in the array section TAU(:KRANK). Other elements of TAU are set to zero on exit.

See Further Details.

The size of TAU must verify:

- size(TAU) = min(size(MAT,1) , size(MAT,2)).

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian matrix in the randomized partial QR algorithm.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to PARTIAL_RQR_CMP_FIXED_PRECISION.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

BLK_SIZE (INPUT, OPTIONAL) integer(i4b) On entry, the block size used in the randomized partial QR factorization.

BLK_SIZE must be greater or equal to one and less than min(m,n) and must be set to a much smaller value than min(m,n) usually, depending also on the architecture of the computer.

By default, BLK_SIZE is set to min(BLKSZ_QR, min(m,n)), where parameter BLKSZ_QR is the default block size for QR related algorithms specified in module Select_Parameters.

NOVER (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the randomized partial QR algorithm.

NOVER must be positive or null and verifies the relationship:

NOVER + BLK_SIZE <= size(MAT,1)

and is adjusted if necessary to verify this relationship in all cases.

See Further Details and the cited references for the meaning and usefulness of the oversampling size in the partial randomized QR algorithm.

By default, the oversampling size is set to 10.

Further Details

The matrix Q is represented as a product of elementary reflectors

Q = H(1) * H(2) * . . . * H(krank), where krank <= min(m , n).

Each H(i) has the form

H(i) = I + beta * (v * v') ,

where beta is a real scalar and v is a real m-element vector with v(1:i-1) = 0. v(i:m) is stored on exit in MAT(i:m,i) and beta in BETA(i).

On exit of PARTIAL_RQR_CMP_FIXED_PRECISION, the matrix Q can be computed explicitly by a call to subroutine ORTHO_GEN_QR with arguments MAT and BETA(:KRANK).

The matrix P is represented in the array IP as follows: If IP(j) = i then the jth column of P is the ith canonical unit vector.

On exit, if the optional argument TAU is absent:

- The elements above the diagonal of the array MAT(:krank,:krank) contain the corresponding elements of the triangular matrix R1.
- The elements of the diagonal of R1 are stored in the array DIAGR.
- The submatrix R2 is stored in MAT(:krank,krank+1:n).
- krank is stored in the real argument KRANK.

If TAU is present, a "partial" complete orthogonal factorization of MAT is computed. The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), which is used to introduce zeros into the kth row of R (e.g., in R2), is given in the form

[I 0]

[0 T(k)]

where

T(k) = I + tau * (u(k) * u(k)') and u(k)' = (1 0 z(k))

tau is a scalar, u(k) is a n-k+1 vector and z(k) is an (n-krank) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of R2.

The Z n-by-n orthogonal matrix is given by

Z = Z(1) * Z(2) * . . . * Z(krank)

On exit, if the optional argument TAU is present:

- The scalar tau defining T(k) is returned in the kth element of TAU and the vector u(k) in the kth row of MAT, such that the elements of z(k) are in MAT(k,krank+1:n). The other elements of u(k) are not stored.
- The elements above the diagonal of the array section MAT(:krank,:krank) contain the corresponding elements of the triangular matrix T1. The elements of the diagonal of T1 are stored in the array DIAGR.
- krank is stored in the real argument KRANK.

In both cases, norm(MAT(KRANK+1:m,KRANK+1:n)) gives the error of the associated partial matrix approximation in the Frobenius norm, and argument RELERR stores the relative error in the Frobenius norm of the matrix approximation on exit.

The computations are parallelized if OPENMP is used. Note also that PARTIAL_RQR_CMP_FIXED_PRECISION uses a randomized "BLAS3" algorithm described in the references (3), (4), (5) and (6), which has about the same efficiency of a "BLAS3" QR algorithm without column pivoting and is thus highly efficient on large matrices.

For further details, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Duersch, J.A., and Gu, M., 2017:** Randomized QR with column pivoting. SIAM J. Sci. Comput., Volume 39, Issue 4, C263-C291.

(4) **Martinsson, P.G., Quintana-Orti, G., Heavner, N., and Van de Geijn, R., 2017:** Householder QR factorization with randomization for column pivoting (HQRRP). SIAM J. Sci. Comput., Volume 39, Issue 2, C96-C115.

(5) **Xiao, J., Gu, M., and Langou, J., 2017:** Fast parallel randomized QR with column pivoting algorithms for reliable low-rank matrix approximations. IEEE 24th International Conference on High Performance Computing (HiPC), IEEE, 2017, 233-242.

(6) **Duersch, J.A., and Gu, M., 2020:** Randomized projection for rank-revealing matrix factorizations and low-rank approximations. SIAM Review, Volume 62, Issue 3, 661-682.

6.17.54 subroutine rqb_cmp (mat, q, b, niter, rng_alg, ortho, comp_qr, ip, tol, tau)

Purpose

RQB_CMP computes a partial QB, QR (eventually with column pivoting) or complete orthogonal factorization of a full m-by-n real matrix MAT using randomized power or subspace iterations.

nqb is the target rank of the partial QB, QR or complete orthogonal decomposition, which is sought, and is equal to the number of columns of the output real matrix argument Q, i.e., nqb = size(Q, 2).

The routine first computes a partial QB factorization of MAT with the help of a randomized algorithm:

MAT = Q * B

, where Q is a m-by-nqb orthonormal matrix, B is a nqb-by-n matrix and the product Q*B is a good approximation of MAT according to the spectral or Frobenius norm.

In a second step:

- if the optional logical argument COMP_QR is used with the value true and the optional array arguments IP and TAU are absent, a QR factorization of B is computed to obtain an approximate QR factorization of MAT:

  MAT = Q * B = Q * (O * R) = (Q * O) * R

  , where O is an nqb-by-nqb orthogonal matrix and R is an nqb-by-n upper trapezoidal matrix.

- if the optional array argument IP is present, a QR factorization of B with column pivoting is performed to obtain an approximate QR factorization with column pivoting of MAT :

  MAT * P = Q * (O * R) = (Q * O) * R

, where P is an n-by-n permutation matrix, O is an nqb-by-nqb orthogonal matrix and R is an nqb-by-n upper trapezoidal matrix.

- if the optional array argument TAU is present, a complete orthogonal factorization of B is performed to obtain an approximate complete orthogonal factorization of MAT:

  MAT = Q * (O * T * Z) = (Q * O) * T * Z

  , where O is an nqb-by-nqb orthogonal matrix, Z is a n-by-n orthogonal matrix and T is a nqb-by-n matrix, which has the form:

  [T1 0]

  , where T1 is a nqb-by-nqb upper triangular matrix.

- if, finally, both the optional array arguments IP and TAU are present, a complete orthogonal factorization of B with column pivoting is performed to obtain an approximate complete orthogonal factorization with column pivoting of MAT :

  MAT * P = Q * (O * T * Z) = (Q * O) * T * Z

  , where P is a n-by-n permutation matrix, O is an nqb-by-nqb orthogonal matrix, Z is a n-by-n orthogonal matrix and T a nqb-by-n matrix, which has the form:

  [T1 0]

  , where T1 is a nqb-by-nqb upper triangular matrix.

If a partial QR or complete orthogonal factorization is computed, we have the following partition of R:

R = [R1 R2]

where R1 is a nqb-by-nqb upper triangular matrix and R2 is a full nqb-by-(n-nqb) matrix. Then, if:

- the optional scalar argument TOL is present and is in]0,1[, the rank of R1 is determined by finding the submatrix of R1 which is defined as the largest leading submatrix whose estimated condition number, in the 1-norm, is less than 1/TOL. The order of this submatrix, krank, is the effective rank of R1.
- the optional scalar argument TOL is present and is equal to 0, the numerical rank of R1, krank, is determined.
- the optional scalar argument TOL is absent or outside [0,1[, the numerical rank of R1, krank, is determined by finding the largest leading submatrix of R1 such that abs(R1[j,j])>0.

In all cases, if krank < nqb, this indicates that column pivoting must be used or, if column pivoting has been already specified, that the rank of MAT is probably less than nqb and nqb = size(Q, 2) has been set to a too large value. In such cases, the subroutine will exit with an error message.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

MAT is not modified by the routine.

Q (OUTPUT) real(stnd), dimension(:,:) On exit, the computed m-by-nqb orthonormal matrix of the partial QB, QR or complete orthogonal factorization of MAT.

See Further Details.

The shape of Q must verify:

- size(Q, 1) = m = size(MAT, 1),
- size(Q, 2) = nqb <= min(size(MAT,1) , size(MAT,2)) .

B (OUTPUT) real(stnd), dimension(:,:) On exit:

- the computed B matrix of the partial QB factorization of MAT if the optional arguments COMP_QR, IP and TAU are absent or if COMP_QR is used with the value false;
- the upper trapezoidal matrix R of the partial QR factorization if the optional logical argument COMP_QR is used with the value true or if the optional array argument IP is present;
- the computed matrices T1 and Z of the partial complete orthogonal factorization of MAT if the optional argument TAU is present.

See Further Details.

The shape of B must verify:

- size(B, 1) = size(Q, 2) = nqb ,
- size(B, 2) = size(MAT, 2) = n .

NITER (INPUT, OPTIONAL) integer(i4b) The number of randomized power or subspace iterations performed in the first phase of the randomized algorithm for computing the preliminary QB factorization.

NITER must be positive or null.

By default, 5 randomized power or subspace iterations are performed.

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian test matrix in the randomized partial QB or QR algorithm.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to RQB_CMP.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, orthonormalization is carried out between each step of the power iterations, to avoid loss of accuracy due to rounding errors. This means that subspace iterations are used instead of power iterations in the QB phase of the algorithm,
- ORTHO=false, orthonormalization is not performed.

The default is to use orthonormalization, e.g., ORTHO=true.

COMP_QR (INPUT, OPTIONAL) logical(lgl) The optional logical argument COMP_QR determines if a partial QB or QR factorization is computed.

On entry, if:

- COMP_QR=true, a partial QR factorization is computed;
- COMP_QR=false, a partial QB factorization is computed.

The default is to compute a partial QB factorization, e.g., COMP_QR=false.

IP (OUTPUT, OPTIONAL) integer(i4b), dimension(:) On entry, if IP is present a partial QR factorization with column pivoting of MAT is performed instead of a QB factorization (e.g., this implies COMP_QR=true).

On exit, if IP(j)=k, then the j-th column of MAT*P was the k-th column of MAT.

See Further Details.

The size of IP must verify: size(MAT, 2) = n.

TOL (INPUT/OUTPUT, OPTIONAL) real(stnd) If COMP_QR=true and TOL is present and is in [0,1[, then:

- the calculations to determine the condition number of of R1 in the 1-norm are performed. Then, TOL is used to determine the effective rank of R1, which is defined as the order of the largest leading triangular submatrix of R1, whose estimated condition number is less than 1/TOL.
- if TOL=0 is specified the numerical rank of R1 is determined.
- on exit, the reciprocal of the condition number is returned in TOL.

If COMP_QR=true, but TOL is not specified or is outside [0,1[:

- the calculations to determine the condition number of R1 are not performed and crude tests on R1(j,j) are done to determine the rank of R1. If TOL is present, it is not changed.

If each element of MAT is correct to, say, 5 digits then TOL=0.00001 should be used on entry.

TAU (OUTPUT, OPTIONAL) real(stnd), dimension(:) On entry, if TAU is present, a complete orthogonal factorization of MAT is computed instead of a QB factorization (e.g., this implies COMP_QR=true).

On exit, the scalars factors of the elementary reflectors defining Z.

See Further Details.

The size of TAU must verify: size(TAU) = nqb = size(Q, 2).

Further Details

For a good introduction to randomized linear algebra, see the references (1) and (2).

The randomized power or subspace iteration was proposed in (3; see Algorithm 4.4) to compute an orthonormal matrix whose range approximates the range of MAT. An approximate partial QB, QR or complete orthogonal factorization can then be computed using the aforementioned orthonormal matrix, see the references (1) and (3) for details.

The orthonormal m-by-nqb matrix Q of the partial QB, QR or complete orthogonal factorization of MAT is computed explicitly and stored on exit in the real array argument Q.

The nqb-by-n matrix B of the partial QB factorization or the upper trapezoidal matrix R of the partial QR or orthogonal factorization of MAT is stored on exit in the real array argument B.

If the integer array argument IP is present, a (partial) QR factorization of MAT with column pivoting is computed, as described above, and on exit the permutation matrix P is represented in the array IP as follows: If IP(j) = i then the jth column of P is the ith canonical unit vector.

Next, if the real array argument TAU is present, a (partial) complete orthogonal factorization of MAT is computed from the partial QR factorization of MAT. The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), which is used to introduce zeros into the kth row of R of the (partial) QR factorization of MAT, is given in the form

[I 0]

[0 T(k)]

where

T(k) = I + tau * (u(k) * u(k)') and u(k)' = (1 0 z(k))

tau is a scalar, u(k) is a n-k+1 vector and z(k) is an (n-nqb) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of R12.

The Z n-by-n orthogonal matrix is finally given by the product

Z = Z(1) * Z(2) * ... * Z(nqb)

On exit:

- the orthogonal matrix Z is stored in factored form. The scalar tau, which defines Z(k) is returned in the kth element of TAU and the vector u(k) in the kth row of B, such that the elements of z(k) are in B(k,nqb+1:n).
- the upper triangular nqb-by-nqb matrix T1, which defines the T factor of the (partial) complete orthogonal factorization of MAT (see above) is stored in the upper triangle of the real array argument B.

Finally, if the optional arguments IP and TAU are both present, a (partial) complete orthogonal factorization with column pivoting of MAT is computed and on exit this factorization is stored with the same conventions as described above.

For further details on randomized linear algebra, computing low-rank matrix approximations using randomized power or subspace iterations, see:

(1) **Martinsson, P.G., 2019:** Randomized methods for matrix computations. arXiv.1607.01649

(2) **Erichson, N.B., Voronin, S., Brunton, S.L., and Kutz, J.N., 2019:** Randomized matrix decompositions using R. arXiv.1608.02148

(3) **Halko, N., Martinsson, P.G., and Tropp, J.A., 2011:** Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions. SIAM Rev., 53, 217-288.

(4) **Gu, M., 2015:** Subspace iteration randomization and singular value problems. SIAM J. Sci. Comput., 37, A1139-A1173.

6.17.55 subroutine rqb_cmp_fixed_precision (mat, relerr, q, b, failure_relerr, niter, rng_alg, blk_size, maxiter_qb, ortho, reortho, niter_qb, comp_qr, ip, tol, tau)

Purpose

RQB_CMP_FIXED_PRECISION computes a partial QB, QR (eventually with column pivoting) or complete orthogonal factorization of a full m-by-n real matrix MAT using randomized power or subspace

iterations.

nqb is the target rank of the partial QB, QR or complete orthogonal factorization, which is sought, and this partial factorization must have an approximation error which fulfills:

|| MAT - Q*B ||_F <= ||MAT||_F * relerr

, where Q is a m-by-nqb orthonormal matrix, B is a nqb-by-n matrix and the matrix product Q*B is the computed matrix approximation. || ||_F is the Frobenius norm and relerr is a prescribed accuracy tolerance for the relative error of the computed matrix approximation, specified in the input argument RELERR.

In other words, the rank, nqb, of the matrix approximation is not known in advance and is determined in the subroutine. This explains why the output array arguments Q and B, which contain the computed factors of the matrix approximation, must be declared in the calling program as pointers.

On exit, nqb is equal to the numbers of columns of the output array pointer argument Q, which is also equal to the numbers of rows of the output array pointer argument B. In other words, nqb = size(Q, 2) = size(B, 1) and the relative error, in the Frobenius norm, of the computed matrix approximation Q * B is output in argument RELERR.

RQB_CMP_FIXED_PRECISION searches incrementally the best (e.g., of smallest rank) Q * B approximation, which fulfills the prescribed accuracy tolerance for the relative error. More precisely, the rank of the matrix approximation is increased progressively of BLK_SIZE by BLK_SIZE until the prescribed accuracy tolerance is satisfied and than adjusted precisely to obtain the Q * B matrix approximation of smallest rank, which satisfies the prescribed tolerance.

Note that the product of the two integer arguments BLK_SIZE and MAXITER_QB (see below for their precise meaning), BLK_SIZE*MAXITER_QB, determines the maximum allowable rank of the matrix approximation, which is sought. In other words, the subroutine will stop the search for the best (e.g., smallest) matrix approximation, which fulfills the requested tolerance, if the rank of this matrix approximation exceeds BLK_SIZE*MAXITER_QB. In that case, the subroutine will return the current matrix approximation (with a rank equal to BLK_SIZE*MAXITER_QB).

In all cases the relative error of the computed matrix approximation is output in argument RELERR.

If, finally, the optional logical argument FAILURE_RELERR is used, it will be set to true if the computed matrix approximation does not fulfill the requested relative error specified on entry in the argument RELERR and to false otherwise.

In a second step:

- if the optional logical argument COMP_QR is used with the value true and the optional array arguments IP and TAU are absent, a QR factorization of B is computed to obtain an approximate QR factorization of MAT:

  MAT = Q * B = Q * (O * R) = (Q * O) * R

  , where O is an nqb-by-nqb orthogonal matrix and R is an nqb-by-n upper trapezoidal matrix.

- if the optional array argument IP is present, a QR factorization of B with column pivoting is performed to obtain an approximate QR factorization with column pivoting of MAT :

  MAT * P = Q * (O * R) = (Q * O) * R

  , where P is an n-by-n permutation matrix, O is an nqb-by-nqb orthogonal matrix and R is an nqb-by-n upper trapezoidal matrix.

- if the optional array argument TAU is present, a complete orthogonal factorization of B is performed to obtain an approximate complete orthogonal factorization of MAT:

  MAT = Q * (O * T * Z) = (Q * O) * T * Z

, where O is an nqb-by-nqb orthogonal matrix, Z is a n-by-n orthogonal matrix and T is a nqb-by-n matrix, which has the form:

[T1 0]

, where T1 is a nqb-by-nqb upper triangular matrix.

- if, finally, both the optional array arguments IP and TAU are present, a complete orthogonal factorization of B with column pivoting is performed to obtain an approximate complete orthogonal factorization with column pivoting of MAT :

  MAT * P = Q * (O * T * Z) = (Q * O) * T * Z

  , where P is a n-by-n permutation matrix, O is an nqb-by-nqb orthogonal matrix, Z is a n-by-n orthogonal matrix and T a nqb-by-n matrix, which has the form:

  [T1 0]

  , where T1 is a nqb-by-nqb upper triangular matrix.

If a partial QR or complete orthogonal factorization is computed, we have the following partition of R:

R = [R1 R2]

where R1 is a nqb-by-nqb upper triangular matrix and R2 is a full nqb-by-(n-nqb) matrix. Then, if:

- the optional scalar argument TOL is present and is in]0,1[, the rank of R1 is determined by finding the submatrix of R1 which is defined as the largest leading submatrix whose estimated condition number, in the 1-norm, is less than 1/TOL. The order of this submatrix, krank, is the effective rank of R1.
- the optional scalar argument TOL is present and is equal to 0, the numerical rank of R1, krank, is determined.
- the optional scalar argument TOL is absent or outside [0,1[, the numerical rank of R1, krank, is determined by finding the largest leading submatrix of R1 such that abs(R1[j,j])>0.

In all cases, if krank < nqb, this indicates that column pivoting must be used or, if column pivoting has been already specified, that the rank of MAT is probably less than nqb and nqb = size(Q, 2) has been set to a too large value. In such cases, the subroutine will exit with an error message.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

MAT is not modified by the routine.

RELERR (INPUT/OUTPUT) real(stnd) On entry, the requested accuracy tolerance for the relative error of the computed matrix approximation.

The preset value for RELERR must be greater than 4*epsilon(RELERR), less than one and verifies:

- RELERR >= 2 * sqrt(epsilon(RELERR)/RELERR)

and is forced to be greater than 2*sqrt(epsilon(RELERR)/RELERR) if this is not the case to avoid loss of accuracy in the algorithm. See reference (6) for more details.

On exit, RELERR contains the relative error of the computed matrix approximation:

- RELERR = ||MAT-Q*B||_F / ||MAT||_F

Q (OUTPUT) real(stnd), dimension(:,:), pointer On exit, the computed m-by-nqb orthonormal matrix of the partial QB, QR or complete orthogonal factorization of MAT.

The statut of the pointer Q must not be undefined on entry. If, on entry, the pointer Q is already allocated, it will be first deallocated and then reallocated with the correct shape.

On exit, the shape of the pointer Q will verify:

- size(Q, 1) = m = size(MAT, 1),
- size(Q, 2) = nqb <= min(size(MAT,1) , size(MAT,2)) .

B (OUTPUT) real(stnd), dimension(:,:), pointer On exit:

- the computed B matrix of the partial QB factorization of MAT if the optional arguments COMP_QR, IP and TAU are absent or if COMP_QR is used with the value false;
- the upper trapezoidal matrix R of the partial QR factorization if the optional logical argument COMP_QR is used with the value true or if the optional array argument IP is present;
- the computed matrices T1 and Z of the partial complete orthogonal factorization of MAT if the optional argument TAU is present.

The statut of the pointer B must not be undefined on entry. If, on entry, the pointer B is already allocated, it will be first deallocated and then reallocated with the correct shape.

On exit, the shape of the pointer B will verify:

- size(B, 1) = size(Q, 2) = nqb ,
- size(B, 2) = size(MAT, 2) = n .

FAILURE_RELERR (OUTPUT, OPTIONAL) logical(lgl) On exit, if the optional logical argument FAILURE_RELERR is present, it is set as follows:

- FAILURE_RELERR = false : indicates successful exit and the computed matrix approximation fulfills the requested relative error specified on entry in the argument RELERR,
- FAILURE_RELERR = true : indicates that the computed matrix approximation has a relative error larger than the requested relative error. This means that the requested accuracy tolerance for the relative error is too small (i.e., RELERR < 2 * sqrt(epsilon(RELERR)/RELERR) or that the input parameters BLK_SIZE and/or MAXITER_QB have a too small value, given the distribution of the singular values of MAT, and must be increased to fullfill the preset accuracy tolerance for the relative error of the matrix approximation.

NITER (INPUT, OPTIONAL) integer(i4b) The number of randomized power or subspace iterations performed in the first phase of the randomized algorithm for computing the preliminary QB factorization.

NITER must be positive or null.

By default, 1 randomized power or subspace iteration is performed.

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian test matrix in the randomized partial QB or QR algorithm.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;

- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to RQB_CMP_FIXED_PRECISION.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

BLK_SIZE (INPUT, OPTIONAL) integer(i4b) On entry, the block size used in the randomized QB factorization, which is used in the first phase of the subroutine.

BLK_SIZE must be greater or equal to one and less than min(m,n) and must be set to a much smaller value than min(m,n) usually, depending also on the architecture of the computer.

By default, BLK_SIZE is set to min(10, min(m,n)).

MAXITER_QB (INPUT, OPTIONAL) integer(i4b) MAXITER_QB controls the maximum number of allowed iterations in the randomized QB algorithm, which is used in the first phase of the subroutine.

MAXITER_QB must be set greater or equal to one and less than int(min(m,n)/BLK_SIZE).

By default, MAXITER_QB is set to max(1, int(min(m,n)/(4*BLK_SIZE))).

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, orthonormalization is carried out between each step of the power iterations, to avoid loss of accuracy due to rounding errors. This means that subspace iterations are used instead of power iterations in the QB phase of the algorithm,
- ORTHO=false, orthonormalization is not performed.

The default is to use orthonormalization, e.g., ORTHO=true.

REORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- REORTHO=true, a reorthogonalization step is performed to avoid the loss of orthogonality in the Gram-Schmidt procedure, which is used in the randomized QB factorization;
- REORTHO=false, a reorthogonalization step is not performed in the Gram-Schmidt procedure.

The default is to use a reorthogonalization step, e.g., REORTHO=true.

NITER_QB (INPUT, OPTIONAL) integer(i4b) The number of subspace iterations performed in the last phase of the QB algorithm for improving the initial QB factorization of MAT.

NITER_QB must be greater or equal to 0.

By default, 2 final subspace iterations are performed.

COMP_QR (INPUT, OPTIONAL) logical(lgl) The optional logical argument COMP_QR determines if a partial QB or QR factorization is computed.

On entry, if:

- COMP_QR=true, a partial QR factorization is computed;
- COMP_QR=false, a partial QB factorization is computed.

The default is to compute a partial QB factorization, e.g., COMP_QR=false.

IP (OUTPUT, OPTIONAL) integer(i4b), dimension(:) On entry, if IP is present a partial QR factorization with column pivoting of MAT is performed instead of a QB factorization (e.g., this implies COMP_QR=true).

On exit, if IP(j)=k, then the j-th column of MAT*P was the k-th column of MAT.

See Further Details.

The size of IP must verify: size(MAT, 2) = n.

TOL (INPUT/OUTPUT, OPTIONAL) real(stnd) If COMP_QR=true and TOL is present and is in [0,1[, then:

- the calculations to determine the condition number of of R1 in the 1-norm are performed. Then, TOL is used to determine the effective rank of R1, which is defined as the order of the largest leading triangular submatrix of R1, whose estimated condition number is less than 1/TOL.
- if TOL=0 is specified the numerical rank of R1 is determined.
- on exit, the reciprocal of the condition number is returned in TOL.

If COMP_QR=true, but TOL is not specified or is outside [0,1[:

- the calculations to determine the condition number of R1 are not performed and crude tests on R1(j,j) are done to determine the rank of R1. If TOL is present, it is not changed.

If each element of MAT is correct to, say, 5 digits then TOL=0.00001 should be used on entry.

TAU (OUTPUT, OPTIONAL) real(stnd), dimension(:), pointer On entry, if TAU is present, a complete orthogonal factorization of MAT is computed instead of a QB factorization (e.g., this implies COMP_QR=true).

On exit, the scalars factors of the elementary reflectors defining Z.

See Further Details.

The statut of the pointer TAU must not be undefined on entry. If, on entry, the pointer TAU is already allocated, it will be first deallocated and then reallocated with the correct size.

On exit, the size of the pointer TAU will verify:

- size(TAU) = nqb = size(Q, 2) = size(B, 1).

Further Details

For a good introduction to randomized linear algebra , see the references (1) and (2).

The randomized subspace iteration was proposed in (3; see Algorithm 4.4) to compute an orthonormal matrix whose range approximates the range of MAT. An approximate partial QB, QR or complete orthogonal factorization can then be computed using the aforementioned orthonormal matrix, see the references (1) and (3) for details.

Usually, the problem of low-rank matrix approximation falls into two categories:

- the fixed-rank problem, where the rank parameter nqb is given;
- the fixed-precision problem, where we seek a partial matrix factorization, Q * B, of rank as small as possible such that

  || MAT - Q*B ||_F <= eps

  , where eps is a given accuracy tolerance.

RQB_CMP_FIXED_PRECISION is dedicated to solve the fixed-precision problem. The fixed-rank problem can be solved by subroutine RQB_CMP.

RQB_CMP_FIXED_PRECISION uses an improved version of the "randQB_FP" algorithm described in the reference (6) to solve the fixed-precision problem.

The orthonormal m-by-nqb matrix Q of the partial QB, QR or complete orthogonal factorization of MAT is computed explicitly and stored on exit in the real array pointer Q.

The nqb-by-n matrix B of the partial QB factorization or the upper trapezoidal matrix R of the partial QR or orthogonal factorization of MAT is stored on exit in the real array pointer B.

If the integer array argument IP is present, a (partial) QR factorization of MAT with column pivoting is computed, as described above, and on exit the permutation matrix P is represented in the array IP as follows: If IP(j) = i then the jth column of P is the ith canonical unit vector.

Next, if the real array pointer TAU is present, a (partial) complete orthogonal factorization of MAT is computed from the partial QR factorization of MAT. The factorization is obtained by Householder's method. The kth transformation matrix, Z(k), which is used to introduce zeros into the kth row of R of the (partial) QR factorization of MAT, is given in the form

[I 0]

[0 T(k)]

where

T(k) = I + tau * (u(k) * u(k)') and u(k)' = (1 0 z(k))

tau is a scalar, u(k) is a n-k+1 vector and z(k) is an (n-nqb) element vector. tau and z(k) are chosen to annihilate the elements of the kth row of R12.

The Z n-by-n orthogonal matrix is finally given by the product

Z = Z(1) * Z(2) * ... * Z(nqb)

On exit:

- the orthogonal matrix Z is stored in factored form. The scalar tau, which defines Z(k) is returned in the kth element of TAU and the vector u(k) in the kth row of B, such that the elements of z(k) are in B(k,nqb+1:n).
- the upper triangular nqb-by-nqb matrix T1, which defines the T factor of the (partial) complete orthogonal factorization of MAT (see above) is stored in the upper triangle of the real array argument B.

Finally, if the optional arguments IP and TAU are both present, a (partial) complete orthogonal factorization with column pivoting of MAT is computed and on exit this factorization is stored with the same conventions as described above.

For further details, on randomized linear algebra, computing low-rank matrix approximations using randomized power or subspace iterations or solving the fixed-precision problem, see:

(1) **Martinsson, P.G., 2019:** Randomized methods for matrix computations. arXiv.1607.01649

(2) **Erichson, N.B., Voronin, S., Brunton, S.L., and Kutz, J.N., 2019:** Randomized matrix decompositions using R. arXiv.1608.02148

(3) **Halko, N., Martinsson, P.G., and Tropp, J.A., 2011:** Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions. SIAM Rev., 53, 217-288.

(4) **GU, M., 2015:** Subspace iteration randomization and singular value problems. SIAM J. Sci. Comput., 37, A1139-A1173.

(5) **Martinsson, P.-G., and Voronin, S., 2016:** A randomized blocked algorithm for efficiently computing rank-revealing factorizations of matrices. SIAM J. Sci. Comput., 38:5, S485-S507.

(6) **Yu, W., Gu, Y., and Li, Y., 2018:** Efficient randomized algorithms for the fixed-precision low-rank matrix approximation. SIAM J. Mat. Ana. Appl., 39:3, 1339-1359.

6.17.56 subroutine id_cmp (mat, ip, t, c, v, diagr, beta, rnorm, tol, random_qr, rng_alg, blk_size, nover)

Purpose

ID_CMP computes a (partial) column Interpolative Decomposition (ID) of a m-by-n real matrix MAT.

A column ID factorization of rank krank approximates MAT as:

MAT = C * V

where C is an m-by-krank matrix, which consists of a subset of krank columns of MAT and V is a krank-by-n matrix, which contains a krank-by-krank identity matrix as a submatrix.

Such column ID factorization can be computed with the help of a (randomized) (partial) QR factorization with column pivoting of MAT (see the references (1), (2) and (4)-(7) for details), which is defined as

MAT * P = Q * R

, where P is a n-by-n permutation matrix, R is an upper triangular or trapezoidal (i.e., if n>m) matrix and Q is a m-by-m orthogonal matrix if the QR factorization is complete.

For computing a column ID decomposition of rank, krank, a partial QR factorization with column pivoting of MAT of rank, krank, is sufficient, e.g., the QR decomposition can be stopped when the numbers of computed columns of Q is equal to krank.

This leads implicitly to the following partition of Q:

Q = [Q1 Q2]

where Q1 is an m-by-krank orthonormal matrix and Q2 is m-by-(m-krank) orthonormal matrix orthogonal to Q1, and to the following corresponding partition of R:

[R11 R12]

[R21 R22]

where R11 is a krank-by-krank triangular matrix, R21 is zero by construction, R12 is a full krank-by-(n-krank) matrix and R22 is a full (m-krank)-by-(n-krank) matrix. This leads to the following approximation of MAT:

MAT = Q1 * [R11 R12] * P'

Here, we see that Q1 * R11 equals the first krank columns of A * P, and so we can define the matrix C in the partial ID factorization of MAT as

C = Q1 * R11 = MAT * P(:,:krank)

, then the dominant term Q1 * [R11 R12] in the partial QR decomposition of MAT * P can be written as

Q1 * [R11 R12] = Q1 * R11 * [I T] = C * [I T]

where I is the identity matrix of order krank and T a krank-by-(n-krank) matrix and a solution to the matrix equation:

R11 * T = R12 ,e.g., T = inv(R11) * R12

Thus, we can obtain a column ID factorization of MAT from its partial QR factorization with column pivoting as

MAT = (Q1 * R11) * [I T] * P' = C * V

where V = [I T] * P' is a krank-by-n matrix.

Finally, observe that the error matrix associated with the column ID decomposition of MAT is given by

MAT - C * V = [0 Q2*R22] * P'

and that the Frobenius norm of this error matrix can be computed efficiently as

|| MAT - C * V ||_F = || Q2 * R22 ||_F = || R22 ||_F

krank is the target rank of the column ID, which is sought, and is set to the number of rows of the output array argument T.

If the optional logical argument RANDOM_QR is used with the value true, a fast randomized partial QR factorization with column pivoting is used in the first phase of the ID algorithm.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is overwritten by details of its partial QR factorization with column pivoting.

See description of PARTIAL_QR_CMP subroutine in module QR_Procedures or of PARTIAL_RQR_CMP subroutine in module Random for further details on how the partial QR decomposition is stored compactly in MAT (and arguments IP, DIAGR and BETA) on exit.

Note that, on exit, the Frobenius norm of the error associated with the computed column ID is equal to norm(mat(krank+1:m,krank+1:n)), which is the same as the Frobenius norm of the error associated with the partial QR decomposition with column pivoting (of rank krank) of MAT.

IP (OUTPUT) integer(i4b), dimension(:) On exit, if IP(j)=k, then the j-th column of MAT*P was the k-th column of MAT.

The matrix C in the (column) ID of MAT corresponds to the subset of the columns of MAT with the indices IP(:krank).

See description of PARTIAL_QR_CMP subroutine in module QR_Procedures and PARTIAL_RQR_CMP subroutine in module Random for further details.

The size of IP must be equal to size(MAT, 2) = n

T (OUTPUT) real(stnd), dimension(:,:) On exit, the krank-by-(n-krank) submatrix T = inv(R11) * R12 in the column ID factorization of MAT.

The shape of T must verify:

- size(T, 1) = krank <= min(size(MAT,1) , size(MAT,2)) = min(m,n)
- size(T, 2) = size(MAT, 2) - size(T, 1) = n - krank

C (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, the m-by-krank matrix C in the ID factorization of MAT. Note that C is computed as C = Q * R11 as MAT is overwritten by its partial QR factorization with column pivoting before C can be estimated.

The shape of C must verify:

- size(C, 1) = size(MAT, 1) = m
- size(C, 2) = size(T, 1) = krank

V (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, the krank-by-n matrix V in the ID factorization of MAT.

The shape of V must verify:

- size(V, 1) = size(T, 1) = krank
- size(V, 2) = size(MAT, 2) = n

RNORM (OUTPUT, OPTIONAL) real(stnd) On exit, the Frobenius norm of the error matrix associated with the column ID computed as || R22 ||_F .

DIAGR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the diagonal elements of the matrix R11 in the partial QR factorization with column pivoting of MAT.

See description of PARTIAL_QR_CMP subroutine in module QR_Procedures and PARTIAL_RQR_CMP subroutine in module Random for further details.

The size of DIAGR must verify:

- size(DIAGR) = krank <= min(size(MAT,1) , size(MAT,2))

BETA (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the scalars factors of the elementary reflectors defining Q in the partial QR factorization with column pivoting of MAT.

See description of PARTIAL_QR_CMP subroutine in module QR_Procedures and PARTIAL_RQR_CMP subroutine in module Random for further details.

The size of BETA must verify:

- size(BETA) = krank <= min(size(MAT,1) , size(MAT,2)).

TOL (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, if TOL is present then:

- if TOL is in]0,1[, calculations to determine the condition number of submatrix R11 in the partial QR factorization of MAT are performed. TOL is used to determine the effective pseudo-rank of R11, which is defined as the order of the largest leading triangular submatrix in the partial QR factorization with column pivoting of MAT, whose estimated condition number in the 1-norm is less than 1/TOL. On exit, the reciprocal of the condition number is returned in TOL.
- if TOL=0 is specified, the calculations to determine the condition number of R11 are not performed and crude tests on R(j,j) are done to determine the numerical pseudo-rank of R11. On exit, TOL is not changed.

The TOL argument is useful to check if the matrix R11 in the QR decomposition of MAT, which must be inverted to compute T (in the column ID of MAT), is sufficiently well conditioned to obtain a stable and robust column ID of MAT.

If the computed pseudo-rank of R11 is less than krank = size(T, 1), the subroutine will exit with an error message as a stable solution to the matrix equation R11 * T = R12 cannot be computed.

On the other hand, if TOL is not specified or is outside [0,1[, the calculations to determine the rank of R11 are not performed and this rank is assumed to be equal to krank = size(T, 1).

If it is possible that MAT may not be of full rank then the stability of the column ID decomposition of MAT, which is sought, can be checked by using TOL=relative precision of the elements in MAT. If each element of MAT is correct to, say, 5 digits then TOL=0.00001 should be used.

See description of PARTIAL_QR_CMP subroutine in module QR_Procedures and PARTIAL_RQR_CMP subroutine in module Random for further details.

RANDOM_QR (INPUT, OPTIONAL) logical(lgl) On entry, if RANDOM_QR is used with the value true, a fast randomized partial QR factorization with column pivoting is used in the first phase of the ID algorithm.

By default, RANDOM_QR = false, i.e., a standard (partial) QR factorization with column pivoting is used in the first phase of the ID algorithm.

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian matrix in the randomized (partial) QR phase of the ID algorithm if RANDOM_QR = true.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to ID_CMP subroutine.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

This optional argument has no effect if logical argument RANDOM_QR is set to false or is absent in the call to ID_CMP subroutine.

BLK_SIZE (INPUT, OPTIONAL) integer(i4b) On entry, the block size used in the randomized partial QR phase of the ID algorithm if RANDOM_QR = true.

BLK_SIZE must be greater or equal to one and less than min(m,n) and must be set to a much smaller value than min(m,n) usually, depending also on the architecture of the computer.

By default, BLK_SIZE is set to min(BLKSZ_QR, min(m,n)), where parameter BLKSZ_QR is the default block size for QR related algorithms specified in module Select_Parameters.

See description of subroutine PARTIAL_RQR_CMP in module Random for the meaning of the the block size in the randomized (partial) QR algorithm.

This optional argument has no effect if logical argument RANDOM_QR is set to false or is absent in the call to ID_CMP.

NOVER (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the randomized partial QR phase of the ID algorithm if RANDOM_QR = true.

NOVER must be positive or null and verifies the relationship:

NOVER + BLK_SIZE <= size(MAT,1)

and is adjusted if necessary to verify this relationship in all cases.

See description of subroutine PARTIAL_RQR_CMP in module Random for the meaning and usefulness of the oversampling size in the randomized (partial) QR algorithm.

By default, the oversampling size is set to 10.

This optional argument has no effect if logical argument RANDOM_QR is set to false or is absent in the call to ID_CMP.

Further Details

For further details on the ID decomposition and computing a column ID from a (randomized) partial QR factorization with column pivoting, see:

(1) **Martinsson, P.G., 2019:** Randomized methods for matrix computations. arXiv.1607.01649

(2) **Erichson, N.B., Voronin, S., Brunton, S.L., and Kutz, J.N., 2019:** Randomized matrix decompositions using R. arXiv.1608.02148

(3) **Voronin, S., Martinsson, P.G., 2015:** Rsvdpack: Subroutines for computing partial singular value decompositions via randomized sampling on single core, multi core, and gpu architectures. arXiv.1502.05366

(4) **Voronin, S., Martinsson, P.G., 2017:** Efficient algorithms for cur and interpolative matrix decompositions Adv Comput Math, Volume 43, 495-516.

(5) **Xiao, J., Gu, M., and Langou, J., 2017:** Fast parallel randomized QR with column pivoting algorithms for reliable low-rank matrix approximations. IEEE 24th International Conference on High Performance Computing (HiPC), IEEE, 2017, 233-242.

(6) **Stewart, G.W., 1999:** Four algorithms for the the efficient computation of truncated pivoted qr approximations to a sparse matrix. Numerische Mathematik, Volume 83, 313-323.

(7) **Berry, M.W., Pulatova, S.A., and Stewart, G.W., 2005:** Algorithm 844: Computing sparse reduced-rank approximations to sparse matrices. ACM Transactions on Mathematical Software, Volume 31, No. 2, 252-269.

6.17.57 subroutine ts_id_cmp (mat, ip_row, ip_col, w, v, skelmat, diagr, beta, rnorm, tol, random_qr, rng_alg, blk_size, nover)

Purpose

TS_ID_CMP computes a (partial) two-sided Interpolative Decomposition (tsID) of a full m-by-n real matrix MAT.

A tsID factorization of rank krank approximates MAT as the matrix product

MAT = W * MAT_skel * V

where W is a m-by-krank matrix, V is a krank-by-n matrix and MAT_skel consists of a squared krank-by-krank submatrix of MAT, which defines the so-called skeleton of MAT.

The tsID factorization can be computed with the help of (randomized) (partial) QR factorizations with column pivoting of MAT and of a matrix derived from its partial QR decomposition, more precisely with a column ID of MAT and a row ID of a subset of the columns of MAT.

The first step is thus to compute a partial column ID decomposition of MAT as

MAT = C * V

where C is a m-by-krank subset of the columns of MAT and V is a krank-by-n matrix. See description of subroutine ID_CMP for more details about the ID decomposition and how such decomposition can be computed from a QR decomposition with column pivoting of MAT.

In a second step, a complete column ID decomposition of C' (e.g., a row ID decomposition of C) is computed as

C' = MAT_skel' * W'

where MAT_skel is a squared krank-by-krank matrix, which is a submatrix of MAT, and W is a m-by-krank matrix. This gives the desired tsID decomposition of MAT as

MAT = W * MAT_skel * V

See the references (1) and (4) and also description of ID_CMP subroutine in module Random for more details on the ID and tsID decompositions.

krank is the target rank of the tsID decomposition, which is sought, and is equal to the number of rows or columns of the output array argument SKELMAT, which stores the skeleton of MAT, e.g., the matrix MAT_skel.

If the optional logical argument RANDOM_QR is used with the value true, fast randomized (partial) QR factorizations with column pivoting are used for computing the column and row ID decompositions in the two phases of the tsID algorithm.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is overwritten by details of its partial QR factorization with column pivoting.

See description of PARTIAL_QR_CMP subroutine in module QR_Procedures or of PARTIAL_RQR_CMP subroutine in module Random for further details on how the partial QR decomposition is stored compactly in MAT (and arguments IP, DIAGR and BETA) on exit.

Note that, the Frobenius norm of the error associated with the computed tsID is the same as the one associated with this partial QR factorization with column pivoting of MAT or its column ID decomposition and is given by norm(mat(krank+1:m,krank+1:n)) on exit. See description of ID_CMP subroutine for further details.

IP_ROW (OUTPUT) integer(i4b), dimension(:) On exit, IP_ROW stores the permutation matrix N in the partial LQ decomposition with row pivoting of C, which is computed to obtain the row ID decomposition of C.

On exit, if IP_ROW(j)=k, then the j-th row of N*C was the k-th row of C, where N = I(IP_ROW,:) and I is the identity matrix of order m.

The matrix MAT_skel in the tsID decomposition of MAT is the submatrix defined as the intersection of the rows IP_ROW(:krank) and the columns IP_COL(:krank) of MAT.

See description of PARTIAL_QR_CMP subroutine in module QR_Procedures and PARTIAL_RQR_CMP and ID_CMP subroutines in module Random for further details.

The size of IP_ROW must be equal to size(MAT, 1) = m.

IP_COL (OUTPUT) integer(i4b), dimension(:) On exit, IP_COL stores the permutation matrix P in the partial QR decomposition with column pivoting of MAT, which is computed to obtain the column ID decomposition of MAT.

If IP_COL(j)=k, then the j-th column of MAT*P was the k-th column of MAT, where P = I(:,IP_COL) and I is the identity matrix of order n.

The matrix C in the (column) ID of MAT corresponds to the subset of the columns of MAT with the indices IP_COL(:krank). Furthermore, the matrix MAT_skel in the tsID decomposition of MAT is the submatrix defined as the intersection of the rows IP_ROW(:krank) and the columns IP_COL(:krank) of MAT.

See description of PARTIAL_QR_CMP subroutine in module QR_Procedures and PARTIAL_RQR_CMP and ID_CMP subroutines in module Random for further details.

The size of IP_COL must be equal to size(MAT, 2) = n.

W (OUTPUT) real(stnd), dimension(:,:) On exit, the m-by-krank matrix W in the tsID factorization of MAT.

The shape of W must verify:

- size(W, 1) = size(MAT, 1) = m
- size(W, 2) = size(SKELMAT, 1) = krank

V (OUTPUT) real(stnd), dimension(:,:) On exit, the krank-by-n matrix V in the tsID factorization of MAT.

The shape of V must verify:

- size(V, 1) = size(SKELMAT, 1) = krank
- size(V, 2) = size(MAT, 2) = n

SKELMAT (OUTPUT) real(stnd), dimension(:,:) On exit, the squared krank-by-krank matrix MAT_skel (e.g., the skeleton of MAT) in the tsID factorization of MAT.

MAT_skel is the submatrix of MAT defines by the rows IP_ROW(:krank) and the columns IP_COL(:krank) of MAT on entry. However, in the routine, MAT_skel is recomputed from the column ID of MAT and row ID of C to save space as MAT is overwritten by details of its partial QR factorization when computing the column ID of MAT.

The shape of SKELMAT must verify:

- size(SKELMAT, 1) = size(SKELMAT, 2) = krank <= min(m,n)

RNORM (OUTPUT, OPTIONAL) real(stnd) On exit, the Frobenius norm of the error matrix associated with the tsID, which is defined as || MAT - W * MAT_skel * V ||_F . Note that this error matrix is the same as the one associated with the partial QR decomposition with column pivoting of MAT or its column ID decomposition.

DIAGR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the diagonal elements of the matrix R11 in the partial QR factorization with column pivoting of MAT.

See description of PARTIAL_QR_CMP subroutine in module QR_Procedures and PARTIAL_RQR_CMP subroutine in module Random for further details.

The size of DIAGR must verify:

- size(DIAGR) = krank <= min(size(MAT,1) , size(MAT,2)) = min(m,n)

BETA (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the scalars factors of the elementary reflectors defining Q in the partial QR factorization with column pivoting of MAT.

See description of PARTIAL_QR_CMP subroutine in module QR_Procedures and PARTIAL_RQR_CMP subroutine in module Random for further details.

The size of BETA must verify:

- size(BETA) = krank <= min(size(MAT,1) , size(MAT,2)) = min(m,n)

TOL (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, if TOL is present then:

- if TOL is in]0,1[, calculations to determine the condition number of submatrix R11 in the partial QR factorization of MAT are performed. TOL is used to determine the effective pseudo-rank of R11, which is defined as the order of the largest leading triangular submatrix in the partial QR factorization with column pivoting of MAT, whose estimated condition number in the 1-norm is less than 1/TOL. On exit, the reciprocal of the condition number is returned in TOL.
- if TOL=0 is specified, the calculations to determine the condition number of R11 are not performed and crude tests on R(j,j) are done to determine the numerical pseudo-rank of R11. On exit, TOL is not changed.

The TOL argument is useful to check if the matrix R11 in the QR decomposition of MAT, which must be inverted to compute T (in the column ID of MAT), is sufficiently well conditioned to obtain a stable and robust column ID of MAT and, consequently, a robust tsID decomposition.

If the computed pseudo-rank of R11 is less than krank = size(SKELMAT, 1), the subroutine will exit with an error message.

On the other hand, if TOL is not specified or is outside [0,1[, the calculations to determine the rank of R11 are not performed and this rank is assumed to be equal to krank = size(SKELMAT, 1).

If it is possible that MAT may not be of full rank then the stability of the tsID decomposition of MAT, which is sought, can be checked by using TOL=relative precision of the elements in MAT. If each element of MAT is correct to, say, 5 digits then TOL=0.00001 should be used.

See description of PARTIAL_QR_CMP subroutine in module QR_Procedures and PARTIAL_RQR_CMP subroutine in module Random for further details.

RANDOM_QR (INPUT, OPTIONAL) logical(lgl) On entry, if RANDOM_QR is used with the value true, fast randomized (partial) QR factorizations are used in the two phases of the tsID algorithm.

By default, RANDOM_QR = false, i.e., a standard (partial) QR factorization with column pivoting is used in the two phases of the tsID algorithm.

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian matrix in the two randomized (partial) QR phases of the tsID algorithm if RANDOM_QR = true.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to TS_ID_CMP subroutine.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

This optional argument has no effect if logical argument RANDOM_QR is set to false or is absent in the call to TS_ID_CMP subroutine.

BLK_SIZE (INPUT, OPTIONAL) integer(i4b) On entry, the block size used in the randomized partial QR phases of the tsID algorithm if RANDOM_QR = true.

BLK_SIZE must be greater or equal to one and less than min(m,n) and must be set to a much smaller value than min(m,n) usually, depending also on the architecture of the computer.

By default, BLK_SIZE is set to min(BLKSZ_QR, min(m,n)), where parameter BLKSZ_QR is the default block size for QR related algorithms specified in module Select_Parameters.

See description of subroutine PARTIAL_RQR_CMP in module Random for the meaning of the the block size in the randomized (partial) QR algorithm.

This optional argument has no effect if logical argument RANDOM_QR is set to false or is absent in the call to TS_ID_CMP subroutine.

NOVER (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the randomized partial QR phases of the tsID algorithm if RANDOM_QR = true.

NOVER must be positive or null and verifies the relationship:

NOVER + BLK_SIZE <= size(MAT,1)

and is adjusted if necessary to verify this relationship in all cases.

See description of subroutine PARTIAL_RQR_CMP in module Random for the meaning and usefulness of the oversampling size in the randomized (partial) QR algorithm.

By default, the oversampling size is set to 10.

This optional argument has no effect if logical argument RANDOM_QR is set to false or is absent in the call to TS_ID_CMP subroutine.

Further Details

For further details on the tsID and computing the tsID decomposition from the (randomized) partial QR factorizations with column pivoting of a matrix, see:

(1) **Martinsson, P.G., 2019:** Randomized methods for matrix computations. arXiv.1607.01649

(2) **Erichson, N.B., Voronin, S., Brunton, S.L., and Kutz, J.N., 2019:** Randomized matrix decompositions using R. arXiv.1608.02148

(3) **Voronin, S., Martinsson, P.G., 2015:** Rsvdpack: Subroutines for computing partial singular value decompositions via randomized sampling on single core, multi core, and gpu architectures. arXiv.1502.05366

(4) **Voronin, S., Martinsson, P.G., 2017:** Efficient algorithms for cur and interpolative matrix decompositions Adv Comput Math, volume 43, 495-516.

(5) **Stewart, G.W., 1999:** Four algorithms for the the efficient computation of truncated pivoted qr approximations to a sparse matrix. Numerische Mathematik, Volume 83, 313-323.

(6) **Berry, M.W., Pulatova, S.A., and Stewart, G.W., 2005:** Algorithm 844: Computing sparse reduced-rank approximations to sparse matrices. ACM Transactions on Mathematical Software, Volume 31, No. 2, 252-269.

6.17.58 subroutine cur_cmp (mat, ip_row, ip_col, u, c, r, rnorm_row, rnorm_col, tol, random_qr, rng_alg, blk_size, nover)

Purpose

CUR_CMP computes a (partial) CUR decomposition of a m-by-n real matrix MAT. A CUR decomposition provides a reduced-rank approximate decomposition of a full m-by-n real matrix MAT of the form:

MAT = C * U * R

where C and R are m-by-krank and krank-by-n matrices, which are, respectively, subsets of the columns and rows of MAT, and U is a krank-by-krank matrix, which is estimated to make the matrix product C * U * R a good approximation of MAT according to the Frobenius norm. The CUR factorization is an important tool for handling large-scale data sets, offering several advantages over the Singular Value Decomposition (SVD): the columns and rows that comprise C and R are representative of the data and they are sparse if MAT is sparse. See references (1) and (2) for a discussion.

Computing an approximate CUR decomposition is generally a three steps process.

The C and R submatrices in the CUR factorization can be first estimated with the help of (randomized) partial QR factorizations with column pivoting of MAT and MAT', respectively.

The initial step is thus to compute a (partial) QR factorization with column pivoting of MAT, which is defined as

MAT * P = Q * T

, where P is an n-by-n permutation matrix, T is an upper triangular or trapezoidal (i.e., if n>m) matrix and Q is a m-by-m orthogonal matrix if the QR factorization is complete.

For computing a CUR decomposition of rank, krank, a partial QR factorization with column pivoting of MAT of rank, krank, is sufficient, e.g., the QR decomposition can be stopped when the numbers of columns of Q is equal to krank .

This leads implicitly to the following partition of Q:

Q = [Q1 Q2]

where Q1 is an m-by-krank orthonormal matrix and Q2 is m-by-(m-krank) orthonormal matrix orthogonal to Q1, and to the following corresponding partition of T:

[T11 T12]

[T21 T22]

where T11 is a krank-by-krank triangular matrix, T21 is zero by construction, T12 is a full krank-by-(n-krank) matrix and T22 is a full (m-krank)-by-(n-krank) matrix. This leads to the following (partial QR) approximation of MAT:

MAT = Q1 * [T11 T12] * P'

Here, we see that Q1 * T11 equals the first krank columns of A * P, and we can define the matrix C in the CUR factorization of MAT as

C = Q1 * T11 = MAT * P(:,:krank)

This choice leads to a subset of the columns of MAT, which will minimize the Frobenius norm, RNORM_COL, of the error associated with the reduced-rank QR approximation or column ID of MAT and, also, of the resulting CUR approximation of MAT as we will illustrate below.

Furthermore, remember that C is also the selected subset of the columns of MAT, which defines the left factor in the column ID of MAT of rank krank:

MAT = (Q1 * T11) * V = C * V

where V = [I Z] * P' is a krank-by-n matrix, I is a krank-by-krank identity matrix and Z is a krank-by-(n-krank) matrix, which is the solution to the matrix equation:

T11 * Z = T12 ,e.g., Z = inv(T11) * T12

see description of ID_CMP subroutine for details.

In a second step, the same partial QR algorithm is applied to MAT' to get a representative subset of the rows of MAT, R, such that

R = L11 * K1 = N(:krank,:) * MAT

where R is a krank-by-n matrix, which is a subset of the rows of MAT, L11 is a krank-by-krank lower triangular matrix, K1 is a krank-by-n matrix with orthonormal rows and N is a m-by-m permutation matrix. Again, this choice leads to a subset of the rows of MAT, R, which will minimize the Frobenius norm, RNORM_ROW, of the error associated with the reduced-rank LQ approximation of MAT (and also of the resulting CUR approximation of MAT).

In a final step, we then seek a krank-by-krank matrix U such that

|| MAT - C * U * R ||_F = min

Once C and U are fixed, it can be shown that the minimum is attained for a matrix U, which is defined as

U = pseudo-inv(C) * MAT * pseudo-inv(R) = inv(T11) * Q1' * MAT * K1' * inv(L11)

where pseudo-inv(C) is the pseudo-inverse of C, which is equal to inv(T11) * Q1' if T11 is of full rank and with similar results for the matrix pseudo-inv(R). See reference (5) for details.

However, such direct computation of U involves the explicit inversion of two triangular matrices, which can be severely affected by ill-conditioning of the matrices T11 and L11. Using the column ID of MAT, we have

Q1' * MAT = T11 * V

, we deduce that

U = inv(T11) * Q1' * MAT * K1' * inv(L11) = V * K1' * inv(L11) = V * pseudo-inv(R)

and we observe that we can also determine U as the solution of the least squares problem:

U * R = V

Such a least squares problem must have an accurate solution as both the rows of R and V should span roughly the same space, namely, the space spanned by the krank leading right singular vectors of MAT (see reference (4) for discussion). If it is the case, the Frobenius norm of the error associated with the resulting CUR decomposition will be about of the same order of the error associated with the column ID of MAT, only slightly larger.

Using this approach, the matrix U can thus be estimated by solving the linear least squares problem:

U * R = V

and this computation never involves the explicit inversion of the matrices T11 and L11, but rather solving two linear least squares problems associated with these two triangular matrices.

Moreover, we have:

|| MAT - C * U * R ||_F <= sqrt(RNORM_COL**2 + RNORM_ROW**2)

See references (5) and (6) for further details.

Note that for matrices whose singular values experience a fast decay, the accuracy of CUR factorization, as computed by CUR_CMP, can deteriorate due to ill-conditioning and the need to solve linear (least squares) problems associated with the triangular matrices T11 and L11 in order to estimate the matrix U. See reference (4) for a more detailed discussion of this problem.

The condition numbers (in the 1-norm) and the ranks of the matrices T11 and L11 are estimated if the optional argument TOL is present in order to check the robustness of the computed CUR decomposition.

krank is the target rank of the partial CUR decomposition, which is sought, and is equal to the number of rows or columns of the output array argument U, which stores the factor U in the CUR of MAT.

If the optional logical argument RANDOM_QR is used with the value true, fast randomized (partial) QR factorizations with column pivoting are used for computing the partial column QR and row LQ decompositions of MAT in the first two phases of the CUR algorithm.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

IP_ROW (OUTPUT) integer(i4b), dimension(:) On exit, IP_ROW stores the permutation matrix N in the partial QR decomposition with column pivoting of MAT', which is computed to obtain the row LQ decomposition of MAT.

On exit, if IP_ROW(j)=k, then the j-th row of N*MAT was the k-th row of MAT, where N = I(IP_ROW,:) and I is the identity matrix of order m.

The matrix R in the CUR of MAT corresponds to the subset of the rows of MAT with the indices IP_ROW(:krank).

See description of PARTIAL_QR_CMP subroutine in module QR_Procedures and PARTIAL_RQR_CMP subroutine in module Random for further details.

The size of IP_ROW must be equal to size(MAT, 1) = m.

IP_COL (OUTPUT) integer(i4b), dimension(:) On exit, IP_COL stores the permutation matrix P in the partial QR decomposition with column pivoting of MAT, which is computed to obtain the column QR decomposition of MAT.

If IP_COL(j)=k, then the j-th column of MAT*P was the k-th column of MAT, where P = I(:,IP_COL) and I is the identity matrix of order n.

The matrix C in the CUR of MAT corresponds to the subset of the columns of MAT with the indices IP_COL(:krank).

See description of PARTIAL_QR_CMP subroutine in module QR_Procedures and PARTIAL_RQR_CMP subroutine in module Random for further details.

The size of IP_COL must be equal to size(MAT, 2) = n.

U (OUTPUT) real(stnd), dimension(:,:) On exit, the krank-by-krank matrix U in the approximate CUR factorization of MAT.

The shape of U must verify:

- size(U, 1) = size(U, 2) = krank <= min(size(MAT,1) , size(MAT,2)) = min(m,n)

C (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, the m-by-krank matrix C in the approximate CUR factorization of MAT.

C consists of the subset of the columns of MAT defined by the indices IP_COL(:krank).

The shape of C must verify:

- size(C, 1) = size(MAT, 1) = m
- size(C, 2) = size(U, 2) = krank

R (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, the krank-by-n matrix R in the approximate CUR factorization of MAT.

R consists of the subset of the rows of MAT defined by the indices IP_ROW(:krank).

The shape of R must verify:

- size(R, 1) = size(U, 1) = krank
- size(R, 2) = size(MAT, 2) = n

RNORM_ROW (OUTPUT, OPTIONAL) real(stnd) On exit, the Frobenius norm of the error matrix associated with the row partial LQ decomposition of MAT (e.g., the QR decomposition of MAT'), computed as || L22 ||_F .

RNORM_COL (OUTPUT, OPTIONAL) real(stnd) On exit, the Frobenius norm of the error matrix associated with the column partial QR decomposition of MAT, computed as || T22 ||_F .

TOL (INPUT, OPTIONAL) real(stnd) If TOL is present and is in [0,1[, then calculations to determine the condition number of submatrices T11 and L11 in the partial QR factorizations of MAT and MAT' are performed.

TOL is used to determine the effective pseudo-ranks of T11 and L11, which are defined as the order of the largest leading triangular submatrices in the partial QR factorizations with column pivoting of MAT and MAT' whose estimated condition numbers in the 1-norm are less than 1/TOL. If TOL=0 is specified, the calculations to determine the condition numbers of T11 and L11 are not performed and crude tests on T(j,j) and L(j,j) are done to determine the numerical pseudo-ranks of T11 and L11.

The TOL argument is useful to check if the matrices T11 and L11 are sufficiently well conditioned to obtain a stable and robust CUR decomposition.

If the computed pseudo-ranks of T11 or L11 are less than krank = size(U, 1), the subroutine will exit with an error message.

On the other hand, if TOL is not specified or is outside [0,1[, the calculations to determine the ranks of T11 and L11 are not performed and these ranks are assumed to be equal to krank = size(U, 1).

If it is possible that MAT, T11 and L11 may not be of full rank, then the stability of the CUR decomposition of MAT, which is sought, can be checked by using TOL=relative precision of the elements in MAT. If each element of MAT is correct to, say, 5 digits then TOL=0.00001 should be used.

See description of PARTIAL_QR_CMP subroutine in module QR_Procedures and PARTIAL_RQR_CMP subroutine in module Random for further details.

RANDOM_QR (INPUT, OPTIONAL) logical(lgl) On entry, if RANDOM_QR is used with the value true, fast randomized (partial) QR factorizations are used in the first two phases of the CUR algorithm.

By default, RANDOM_QR = false, i.e., a standard (partial) QR factorization with column pivoting is used in the first two phases of the CUR algorithm.

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian matrix in the two randomized (partial) QR phases of the CUR algorithm if RANDOM_QR = true.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to CUR_CMP subroutine.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

This optional argument has no effect if logical argument RANDOM_QR is set to false or is absent in the call to CUR_CMP subroutine.

BLK_SIZE (INPUT, OPTIONAL) integer(i4b) On entry, the block size used in the randomized partial QR phases of the CUR algorithm if RANDOM_QR = true.

BLK_SIZE must be greater or equal to one and less than min(m,n) and must be set to a much smaller value than min(m,n) usually, depending also on the architecture of the computer.

By default, BLK_SIZE is set to min(BLKSZ_QR, min(m,n)), where parameter BLKSZ_QR is the default block size for QR related algorithms specified in module Select_Parameters.

See description of subroutine PARTIAL_RQR_CMP in module Random for the meaning of the the block size in the randomized (partial) QR algorithm.

This optional argument has no effect if logical argument RANDOM_QR is set to false or is absent in the call to CUR_CMP subroutine.

NOVER (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the randomized partial QR phases of the CUR algorithm if RANDOM_QR = true.

NOVER must be positive or null and verifies the relationship:

NOVER + BLK_SIZE <= size(MAT,1)

and is adjusted if necessary to verify this relationship in all cases.

See description of subroutine PARTIAL_RQR_CMP in module Random for the meaning and usefulness of the oversampling size in the randomized (partial) QR algorithm.

By default, the oversampling size is set to 10.

This optional argument has no effect if logical argument RANDOM_QR is set to false or is absent in the call to CUR_CMP subroutine.

Further Details

For further details on the CUR and computing the CUR decomposition from (randomized) partial QR factorizations with column pivoting of a matrix and its transpose, see:

(1) **Mahoney, M.W., and Drineas, P., 2009:** CUR matrix decompositions for improved data analysis. PNAS, Volume 106, No. 3, 697-702.

(2) **Martinsson, P.G., 2019:** Randomized methods for matrix computations. arXiv.1607.01649

(3) **Voronin, S., Martinsson, P.G., 2015:** Rsvdpack: Subroutines for computing partial singular value decompositions via randomized sampling on single core, multi core, and gpu architectures. arXiv.1502.05366

(4) **Voronin, S., Martinsson, P.G., 2017:** Efficient algorithms for cur and interpolative matrix decompositions Adv Comput Math, Volume 43, 495-516.

(5) **Stewart, G.W., 1999:** Four algorithms for the the efficient computation of truncated pivoted qr approximations to a sparse matrix. Numerische Mathematik, Volume 83, 313-323.

(6) **Berry, M.W., Pulatova, S.A., and Stewart, G.W., 2005:** Algorithm 844: Computing sparse reduced-rank approximations to sparse matrices. ACM Transactions on Mathematical Software, Volume 31, No. 2, 252-269.

6.17.59 `subroutine simple_shuffle ( vec )`

purpose

This subroutine shuffles all the elements of the real vector VEC.

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the real vector to be shuffled.

On exit, the permuted real vector.

Further Details

For more details and algorithm, see:

(1) **Noreen, E.W., 1989:** Computer-intensive methods for testing hypotheses: an introduction. Wiley and Sons, New York, USA, ISBN:978-0-471-61136-3

6.17.60 `subroutine simple_shuffle ( vec )`

purpose

This subroutine shuffles all the elements of the complex vector VEC.

Arguments

VEC (INPUT/OUTPUT) complex(stnd), dimension(:) On entry, the complex vector to be shuffled.

On exit, the permuted complex vector.

Further Details

For more details and algorithm, see:

(1) **Noreen, E.W., 1989:** Computer-intensive methods for testing hypotheses: an introduction. Wiley and Sons, New York, USA, ISBN:978-0-471-61136-3

6.17.61 subroutine simple_shuffle (vec)

purpose

This subroutine shuffles all the elements of the integer vector VEC.

Arguments

VEC (INPUT/OUTPUT) intger(i4b), dimension(:) On entry, the integer vector to be shuffled.

On exit, the permuted integer vector.

Further Details

For more details and algorithm, see:

(1) **Noreen, E.W., 1989:** Computer-intensive methods for testing hypotheses: an introduction. Wiley and Sons, New York, USA, ISBN:978-0-471-61136-3

6.17.62 subroutine drawsample (nsample, pop)

purpose

This subroutine may be used to draw a sample, without replacement of size NSAMPLE from a population of size SIZE(POP). On output, the integer vector POP(1:NSAMPLE) indicates which observations are included in the sample.

The integer vector POP must be dimensioned at least as large as NSAMPLE in the calling program.

Arguments

NSAMPLE (INPUT) intger(i4b) On entry, the size of the sample.

POP (OUTPUT) integer(i4b), dimension(:) On exit, the indices of the observations belonging to the sample are in POP(1:NSAMPLE) and the indices of the observations, which are not in the sample are in POP(NSAMPLE+1:).

The size of POP must greater or equal to NSAMPLE. If this condition is not meet POP(:) is set to -1.

Further Details

For more details and algorithm, see:

(1) **Noreen, E.W., 1989:** Computer-intensive methods for testing hypotheses: an introduction. Wiley and Sons, New York, USA, ISBN:978-0-471-61136-3

6.17.63 subroutine drawbootsample (npop, sample)

purpose

This subroutine may be used to draw a bootstrap random sample of size SIZE(SAMPLE) from a population of size NPOP. On output, the integer vector SAMPLE indicates which observations are included in the bootstrap sample.

Arguments

NPOP (INPUT) intger(i4b) On entry, the size of the population.

SAMPLE (OUTPUT) integer(i4b), dimension(:) On exit, the indices of the observations belonging to the sample.

Further Details

The sampling is done with replacement, meaning that the sample may contain duplicate observations.

For more details and algorithm, see:

(1) **Noreen, E.W., 1989:** Computer-intensive methods for testing hypotheses: an introduction. Wiley and Sons, New York, USA, ISBN:978-0-471-61136-3

6.18 Module_Reals_Constants

Copyright 2022 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

THIS MODULE PROVIDES NAMES FOR ALL REQUIRED LITERAL REAL VALUES OF KIND 'stnd' AND 'extd' USED IN STATPACK.

BY ONLY USING REAL VALUES AS DEFINED WITHIN THIS MODULE, ALL PROBLEMS ASSOCIATED WITH THE PRECISION OF REAL LITERAL VALUES CAN BE TOTALLY AVOIDED.

LATEST REVISION : 22/01/2022

6.19 Module_SVD_Procedures

Copyright 2025 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING SUBROUTINES AND FUNCTIONS FOR COMPUTING FULL, PARTIAL SVD OR QLP DECOMPOSITIONS AND GENERALIZED INVERSE OF A MATRIX.

SUBROUTINES FOR COMPUTING PARTIAL EIGENVALUE, SVD OR QLP DECOMPOSITIONS BASED ON RANDOMIZED ALGORITHMS ARE ALSO PROVIDED.

LATEST REVISION : 15/09/2025

6.19.1 `subroutine bd_cmp ( mat, d, e, tauq, taup )`

Purpose

BD_CMP reduces a general m-by-n matrix MAT to upper or lower bidiagonal form BD by an orthogonal transformation :

Q' * MAT * P = BD

where Q and P are orthogonal. If:

- m >= n, BD is upper bidiagonal;
- m < n, BD is lower bidiagonal.

BD_CMP computes BD, Q and P.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the general m-by-n matrix to be reduced.

On exit, if:

- m >= n, the elements on and below the diagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array TAUP, represent the orthogonal matrix P as a product of elementary reflectors;
- m < n, the elements below the diagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements on and above the diagonal, with the array TAUP, represent the orthogonal matrix P as a product of elementary reflectors.

See Further Details.

D (OUTPUT) real(stnd), dimension(:) The diagonal elements of the bidiagonal matrix BD

The size of D must be min(size(MAT,1) , size(MAT,2)).

E (OUTPUT) real(stnd), dimension(:) The off-diagonal elements of the bidiagonal matrix BD:

- if m >= n, E(i) = BD(i-1,i) for i = 2,3,. . . ,n;
- if m < n, E(i) = BD(i,i-1) for i = 2,3,. . . ,m.

The size of E must be min(size(MAT,1) , size(MAT,2)).

TAUQ (OUTPUT) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix Q. See Further Details.

The size of TAUQ must be min(size(MAT,1) , size(MAT,2)).

TAUP (OUTPUT) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix P. See Further Details.

The size of TAUP must be min(size(MAT,1) , size(MAT,2)).

Further Details

The matrices Q and P are represented as products of elementary reflectors:

If m >= n,

Q = H(1) * H(2) * . . . * H(n) and P = G(1) * G(2) * . . . * G(n-1)

Each H(i) and G(i) has the form:

H(i) = I + tauq * u * u' and G(i) = I + taup * v * v'

where tauq and taup are real scalars, and u and v are real vectors; u(1:i-1) = 0 and u(i:m) is stored on exit in MAT(i:m,i); v(1:i) = 0 and v(i+1:n) is stored on exit in MAT(i,i+1:n); tauq is stored in TAUQ(i) and taup in TAUP(i).

If m < n,

Q = H(1) * H(2) * . . . * H(m-1) and P = G(1) * G(2) * . . . * G(m)

Each H(i) and G(i) has the form:

H(i) = I + tauq * u * u' and G(i) = I + taup * v * v'

where tauq and taup are real scalars, and u and v are real vectors; u(1:i) = 0 and u(i+1:m) is stored on exit in MAT(i+1:m,i); v(1:i-1) = 0 and v(i:n) is stored on exit in MAT(i,i:n); tauq is stored in TAUQ(i) and taup in TAUP(i).

The contents of MAT on exit are illustrated by the following examples:

m = 6 and n = 5 (m >= n):

(u1 v1 v1 v1 v1)

(u1 u2 v2 v2 v2)

(u1 u2 u3 v3 v3)

(u1 u2 u3 u4 v4)

(u1 u2 u3 u4 u5)

(u1 u2 u3 u4 u5)

m = 5 and n = 6 (m < n):

(v1 v1 v1 v1 v1 v1)

(u1 v2 v2 v2 v2 v2)

(u1 u2 v3 v3 v3 v3)

(u1 u2 u3 v4 v4 v4)

(u1 u2 u3 u4 v5 v5)

where ui denotes an element of the vector defining H(i), and vi an element of the vector defining G(i).

This subroutine is adapted from the routine DGEBD2 in LAPACK. An efficient variant of the classic Golub and Kahan Householder bidiagonalization algorithm is used. This variant reduces the traffic on the data bus from four reads and two writes per column-row elimination of the bidiagonalization process to one read and one write. Furthermore, the algorithm is parallelized if OPENMP is used.

For further details on the bidiagonal reduction algorithm and its use or the efficient variant used here, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Howell, G.W., Demmel, J., Fulton, C.T., Hammarling, S., and Marmol, K., 2008:** Cache efficient bidiagonalization using BLAS 2.5 operators. ACM Transactions on Mathematical Software (TOMS) Volume 34, Issue 3.

6.19.2 `subroutine bd_cmp ( mat, d, e, tauq )`

Purpose

BD_CMP reduces a general m-by-n matrix MAT to upper or lower bidiagonal form BD by an orthogonal transformation :

Q' * MAT * P = BD

where Q and P are orthogonal. If:

- m >= n, BD is upper bidiagonal;
- m < n, BD is lower bidiagonal.

BD_CMP computes only BD and Q.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the general m-by-n matrix to be reduced.

On exit, if:

- m >= n, the elements on and below the diagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the diagonal are destroyed;
- m < n, the elements below the diagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements on and above the diagonal are destroyed.

See Further Details.

D (OUTPUT) real(stnd), dimension(:) The diagonal elements of the bidiagonal matrix BD

The size of D must be min(size(MAT,1) , size(MAT,2)).

E (OUTPUT) real(stnd), dimension(:) The off-diagonal elements of the bidiagonal matrix BD:

- if m >= n, E(i) = BD(i-1,i) for i = 2,3,. . . ,n;
- if m < n, E(i) = BD(i,i-1) for i = 2,3,. . . ,m.

The size of E must be min(size(MAT,1) , size(MAT,2)).

TAUQ (OUTPUT) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix Q. See Further Details.

The size of TAUQ must be min(size(MAT,1) , size(MAT,2)).

Further Details

The matrice Q is represented as products of elementary reflectors:

If m >= n,

Q = H(1) * H(2) * . . . * H(n)

Each H(i) has the form:

H(i) = I + tauq * u * u'

where tauq is a real scalar and u is a real vector; u(1:i-1) = 0 and u(i:m) is stored on exit in MAT(i:m,i); tauq is stored in TAUQ(i).

If m < n,

Q = H(1) * H(2) * . . . * H(m-1)

Each H(i) has the form:

H(i) = I + tauq * u * u'

where tauq is a real scalar and u is a real vector; u(1:i) = 0 and u(i+1:m) is stored on exit in MAT(i+1:m,i); tauq is stored in TAUQ(i).

The contents of MAT on exit are illustrated by the following examples:

m = 6 and n = 5 (m > n):

(u1 xx xx xx xx)

(u1 u2 xx xx xx)

(u1 u2 u3 xx xx)

(u1 u2 u3 u4 xx)

(u1 u2 u3 u4 u5)

(u1 u2 u3 u4 u5)

m = 5 and n = 6 (m < n):

(xx xx xx xx xx xx)

(u1 xx xx xx xx xx)

(u1 u2 xx xx xx xx)

(u1 u2 u3 xx xx xx)

(u1 u2 u3 u4 xx xx)

where ui denotes an element of the vector defining H(i). The upper triangular part of MAT is destroyed on exit.

This subroutine is adapted from the routine DGEBD2 in LAPACK. An efficient variant of the classic Golub and Kahan Householder bidiagonalization algorithm is used. This variant reduces the traffic on the data bus from four reads and two writes per column-row elimination of the bidiagonalization process to one read and one write. Furthermore, the algorithm is parallelized if OPENMP is used.

For further details on the bidiagonal reduction algorithm and its use or the efficient variant used here, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Howell, G.W., Demmel, J., Fulton, C.T., Hammarling, S., and Marmol, K., 2008:** Cache efficient bidiagonalization using BLAS 2.5 operators. ACM Transactions on Mathematical Software (TOMS) Volume 34, Issue 3.

6.19.3 `subroutine bd_cmp ( mat, d, e, tauq, taup, rlmat, tauo )`

Purpose

BD_CMP reduces a general m-by-n matrix MAT to upper bidiagonal form BD by a two-step algorithm:

- If m >= n, a QR factorization of the real m-by-n matrix MAT is first computed

  MAT = O * R

where O is orthogonal and R is upper triangular. In a second step, the n-by-n upper triangular matrix R is reduced to upper bidiagonal form BD by an orthogonal transformation :

Q' * R * P = BD

where Q and P are orthogonal and BD is an upper bidiagonal matrix.

- If m < n, an LQ factorization of the real m-by-n matrix MAT is first computed

  MAT = L * O

where O is orthogonal and L is lower triangular. In a second step, the m-by-m lower triangular matrix L is reduced to upper bidiagonal form BD by an orthogonal transformation :

Q' * L * P = BD

where Q and P are orthogonal and BD is an upper bidiagonal matrix.

BD_CMP computes O, BD, Q and P. The matrix O is stored in factored form if the optional argument TAUO is present or explicitly computed if this argument is absent.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the general m-by-n matrix to be reduced.

On exit, if:

- m >= n, the elements on and below the diagonal, with the array TAUO, represent the orthogonal matrix O of the QR factorization of MAT, as a product of elementary reflectors, if the argument TAUO is present. Otherwise, the argument MAT contains the first n columns of O on output.
- m < n, the elements on and above the diagonal, with the array TAUO, represent the orthogonal matrix O of the LQ factorization of MAT, as a product of elementary reflectors, if the argument TAUO is present. Otherwise, the argument MAT contains the first m rows of O on output.

See Further Details.

D (OUTPUT) real(stnd), dimension(:) The diagonal elements of the bidiagonal matrix BD

The size of D must be min(size(MAT,1) , size(MAT,2)).

E (OUTPUT) real(stnd), dimension(:) The off-diagonal elements of the bidiagonal matrix BD:

E(i) = BD(i-1,i) for i = 2,3,. . . ,n;

The size of E must be min(size(MAT,1) , size(MAT,2)).

TAUQ (OUTPUT) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix Q. See Further Details.

The size of TAUQ must be min(size(MAT,1) , size(MAT,2)).

TAUP (OUTPUT) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix P. See Further Details.

The size of TAUP must be min(size(MAT,1) , size(MAT,2)).

RLMAT (OUTPUT) real(stnd), dimension(:,:)

On exit, the elements on and below the diagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array TAUP, represent the orthogonal matrix P as a product of elementary reflectors;

See Further Details.

The shape of RLMAT must verify:

- size(RLMAT, 1) = size(RLMAT, 2) = min(size(MAT,1) , size(MAT,2)).

TAUO (OUTPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix O of the QR or LQ decomposition of MAT.

If the optional argument TAUO is present, the orthogonal matrix O is stored in factored form, as a product of elementary reflectors, in the argument MAT on exit.

If the optional argument TAUO is absent, the orthogonal matrix O is explicitly generated and stored in the argument MAT on exit.

See description of the argument MAT above and Further Details below.

The size of TAUO must be min(size(MAT,1) , size(MAT,2)).

Further Details

If m >= n, the matrix O of the QR factorization of MAT is represented as a product of elementary reflectors

O = W(1) * W(2) * . . . * W(n)

Each W(i) has the form

W(i) = I + tauo * (v * v') ,

where tauo is a real scalar and v is a real m-element vector with v(1:i-1) = 0. v(i:m) is stored on exit in MAT(i:m,i) and tauo in TAUO(i). If the optional argument TAUO is absent, the first n columns of O are generated and stored in the argument MAT.

If m < n, The matrix O of the LQ factorization of MAT is represented as a product of elementary reflectors

O = W(m) * . . . * W(2) * W(1)

Each W(i) has the form

W(i) = I + tauo * (v * v') ,

where tauo is a real scalar and v is a real n-element vector with v(1:i-1) = 0. v(i:n) is stored on exit in MAT(i,i:n) and tauo in TAUO(i).

A blocked algorithm is used for computing the QR or LQ factorization of MAT. Furthermore, the computations are parallelized if OPENMP is used.

After, the initial QR or LQ factorization of MAT, the (upper or lower) triangular matrix is reduced to upper bidiagonal form BD.

The matrices Q and P of the bidiagonal factorization of the triangular matrix R or L are represented as products of elementary reflectors:

Q = H(1) * H(2) * ... * H(k) and P = G(1) * G(2) * ... * G(k-1)

, where k = min(size(MAT,1) , size(MAT,2)). Each H(i) and G(i) has the form:

H(i) = I + tauq * u * u' and G(i) = I + taup * v * v'

where tauq and taup are real scalars, and u and v are real vectors; u(1:i-1) = 0 and u(i:min(m,n)) is stored on exit in RLMAT(i:min(m,n),i); v(1:i) = 0 and v(i+1:min(m,n)) is stored on exit in RLMAT(i,i+1:min(m,n)); tauq is stored in TAUQ(i) and taup in TAUP(i).

The contents of RLMAT on exit are illustrated by the following example:

m = 6 and n = 5 (m >= n):

(u1 v1 v1 v1 v1)

(u1 u2 v2 v2 v2)

(u1 u2 u3 v3 v3)

(u1 u2 u3 u4 v4)

(u1 u2 u3 u4 u5)

where ui denotes an element of the vector defining H(i), and vi an element of the vector defining G(i).

An efficient variant of the classic Golub and Kahan Householder bidiagonalization algorithm is used. This variant reduces the traffic on the data bus from four reads and two writes per column-row elimination of the bidiagonalization process to one read and one write. Furthermore, the algorithm is parallelized if OPENMP is used.

For further details on the bidiagonal reduction algorithm and its use or the variant used here, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Howell, G.W., Demmel, J., Fulton, C.T., Hammarling, S., and Marmol, K., 2008:** Cache efficient bidiagonalization using BLAS 2.5 operators. ACM Transactions on Mathematical Software (TOMS) Volume 34, Issue 3.

6.19.4 subroutine bd_cmp (mat, d, e)

Purpose

BD_CMP reduces a general m-by-n matrix MAT to upper or lower bidiagonal form BD by an orthogonal transformation :

Q' * MAT * P = BD

where Q and P are orthogonal. If:

- m >= n, BD is upper bidiagonal;
- m < n, BD is lower bidiagonal.

BD_CMP computes only BD and the matrices Q and P are not saved.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the general m-by-n matrix to be reduced.

On exit, the general m-by-n matrix is destroyed.

D (OUTPUT) real(stnd), dimension(:) The diagonal elements of the bidiagonal matrix BD

The size of D must be min(size(MAT,1) , size(MAT,2)).

E (OUTPUT) real(stnd), dimension(:) The off-diagonal elements of the bidiagonal matrix BD:

- if m >= n, E(i) = BD(i-1,i) for i = 2,3,. . . ,n;
- if m < n, E(i) = BD(i,i-1) for i = 2,3,. . . ,m.

The size of E must be min(size(MAT,1) , size(MAT,2)).

Further Details

This subroutine is adapted from the routine DGEBD2 in LAPACK. An efficient variant of the classic Golub and Kahan Householder bidiagonalization algorithm is used. This variant reduces the traffic on the data bus from four reads and two writes per column-row elimination of the bidiagonalization process to one read and one write. Furthermore, the algorithm is parallelized if OPENMP is used.

For further details on the bidiagonal reduction algorithm and its use or the efficient variant used here, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Howell, G.W., Demmel, J., Fulton, C.T., Hammarling, S., and Marmol, K., 2008:** Cache efficient bidiagonalization using BLAS 2.5 operators. ACM Transactions on Mathematical Software (TOMS) Volume 34, Issue 3.

6.19.5 `subroutine bd_cmp2 ( mat, d, e, p, failure, gen_p, reortho )`

Purpose

BD_CMP2 reduces a m-by-n matrix MAT with m >= n to upper bidiagonal form BD by an orthogonal transformation :

Q' * MAT * P = BD

where Q and P are orthogonal.

BD_CMP2 computes BD, Q and P using the one-sided Ralha-Barlow bidiagonal reduction algorithm.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the general m-by-n matrix to be reduced.

On exit, the first n columns of Q are stored in in MAT(1:m,1:n).

The shape of MAT must verify: size(MAT, 1) >= size(MAT, 2) = n .

D (OUTPUT) real(stnd), dimension(:) The diagonal elements of the bidiagonal matrix BD

The size of D must be size(MAT, 2) = n .

E (OUTPUT) real(stnd), dimension(:) The off-diagonal elements of the bidiagonal matrix BD:

E(i) = BD(i-1,i) for i = 2,3,. . . ,n;

The size of E must be size(MAT, 2) = n .

P (OUTPUT) real(stnd), dimension(:,:) On exit, the n-by-n matrix P.

The shape of P must verify: size(P, 1) = size(P, 2) = n .

FAILURE (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE = false : indicates successful exit ;
- FAILURE = true : indicates that MAT is nearly singular and some loss of orthogonality of Q can be expected in the Ralha-Barlow one-sided algorithm.

See further details.

GEN_P (INPUT, OPTIONAL) logical(lgl) If the optional argument GEN_P is used and is set to true, the orthogonal matrix P is generated on output of the subroutine. If this argument is set to false, the orthogonal matrix is stored in factored form as products of elementary reflectors in the lower triangle of the array P.

See further details.

The default is GEN_P = true.

REORTHO (INPUT, OPTIONAL) logical(lgl) If the optional argument REORTHO is set to true, re-orthogonalization of the matrix Q is performed (when needed). If this optional logical argument is set to false, reorthogonalization of Q is never performed and the columns of the matrix Q can be far from orthogonal to each other for input matrices MAT with a large condition number.

See further details.

The default is REORTHO = true.

Further Details

This subroutine is an implementation of the Ralha-Barlow one-sided method to reduce a rectangular matrix MAT to bidiagonal form BD. Q is computed by a recurrence relationship and P as a product of n-1 elementary reflectors (e.g. Householder transformations):

P = G(1) * G(2) * . . . * G(n-1)

Each G(i) has the form:

G(i) = I + taup * v * v'

where taup is a real scalar, and v is a real vector. IF GEN_P is used and set to false, the n-1 G(i) elementary reflectors are stored in the lower triangle of the array P. For the G(i) reflector, taup is stored in P(i+1,1) and v is stored in P(i+1:n,i+1). IF GEN_P is set to true, P is generated in P(:n,:n).

In addition, P(1,1) is set to -1 if GEN_P=false and is equal to 1 if GEN_P=true. In other words, the value of P(1,1) indicates if the orthogonal matrix P is stored in factored form or not. Note that if n is equal to 1, no elementary reflectors are needed and consequently P(1,1) is set to 1, independently of the value of GEN_P.

This is the blocked version of the algorithm. See the references (1), (2) and (3) for further details. Note also that the blocked algorithm implemented here is more efficient than the version described in the reference (3). Furthermore the algorithm is parallelized if OPENMP is used.

Since Q is computed by a recurrence relationship, a loss of orthogonality of Q can be observed when the rectangular matrix MAT is singular or nearly singular or has a large condition number, see the reference (2) for details.

To correct partly this deficiency, partial reorthogonalization can be performed to ensure orthogonality at the expense of speed of computation. The reorthogonalization uses the Gram-Schmidt method described in the reference (4) and is activated if the optional logical argument REORTHO is set to true.

The reference (2) also explains how to handle the case of an exactly singular matrix MAT (a very rare event). However, in this subroutine, the partial reorthogonalization described in the reference (4), which is used when REORTHO is set true, corrects automatically this problem.

For further details, see:

(1) **Ralha, R.M.S., 2003:** One-sided reduction to bidiagonal form. Linear Algebra Appl., No 358, pp. 219-238.

(2) **Barlow, J.L., Bosner, N., and Drmac, Z., 2005:** A new stable bidiagonal reduction algorithm. Linear Algebra Appl., No 397, pp. 35-84.

(3) **Bosner, N., and Barlow, J.L., 2007:** Block and Parallel versions of one-sided bidiagonalization. SIAM J. Matrix Anal. Appl., Volume 29, No 3, pp. 927-953.

(4) **Stewart, G.W., 2007:** Block Gram-Schmidt Orthogonalization. Report TR-4823, Department of Computer Science, College Park, University of Maryland.

6.19.6 `subroutine bd_cmp2 ( mat, d, e, failure, reortho )`

Purpose

BD_CMP2 reduces a m-by-n matrix MAT with m >= n to upper bidiagonal form BD by an orthogonal transformation :

Q' * MAT * P = BD

where Q and P are orthogonal.

BD_CMP2 computes BD and Q using the one-sided Ralha-Barlow bidiagonal reduction algorithm.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the general m-by-n matrix to be reduced.

On exit, the first n columns of Q are stored in in MAT(1:m,1:n).

The shape of MAT must verify: size(MAT, 1) >= size(MAT, 2) = n .

D (OUTPUT) real(stnd), dimension(:) The diagonal elements of the bidiagonal matrix BD

The size of D must be size(MAT, 2) = n .

E (OUTPUT) real(stnd), dimension(:) The off-diagonal elements of the bidiagonal matrix BD:

E(i) = BD(i-1,i) for i = 2,3,. . . ,n;

The size of E must be size(MAT, 2) = n .

FAILURE (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE = false : indicates successful exit ;
- FAILURE = true : indicates that MAT is nearly singular and some loss of orthogonality of Q can be expected in the Ralha-Barlow one-sided algorithm.

See further details.

REORTHO (INPUT, OPTIONAL) logical(lgl) If the optional argument REORTHO is set to true, reorthogonalization of the matrix Q is performed (when needed). If this optional logical argument is set to false, reorthogonalization of Q is never performed and the columns of the matrix Q can be far from orthogonal to each other for input matrices MAT with a large condition number.

See further details.

The default is REORTHO = true.

Further Details

This subroutine is an implementation of the Ralha-Barlow one-sided method to reduce a rectangular matrix MAT to bidiagonal form BD. Q is computed by a recurrence relationship.

This is the blocked version of the algorithm. See the references (1), (2) and (3) for further details. Note also that the blocked algorithm implemented here is more efficient than the version described in the reference (3). Furthermore the algorithm is parallelized if OPENMP is used.

Since Q is computed by a recurrence relationship, a loss of orthogonality of Q can be observed when the rectangular matrix MAT is singular or nearly singular or has a large condition number, see the reference (2) for details.

To correct partly this deficiency, partial reorthogonalization can be performed to ensure orthogonality at the expense of speed of computation. The reorthogonalization uses the Gram-Schmidt method described in the reference (4) and is activated if the optional logical argument REORTHO is set to true.

The reference (2) also explains how to handle the case of an exactly singular matrix MAT (a very rare event). However, in this subroutine, the partial reorthogonalization described in the reference (4), which is used when REORTHO is set true, corrects automatically this problem.

For further details, see:

(1) **Ralha, R.M.S., 2003:** One-sided reduction to bidiagonal form. Linear Algebra Appl., No 358, pp. 219-238.

(2) **Barlow, J.L., Bosner, N., and Drmac, Z., 2005:** A new stable bidiagonal reduction algorithm. Linear Algebra Appl., No 397, pp. 35-84.

(3) **Bosner, N., and Barlow, J.L., 2007:** Block and Parallel versions of one-sided bidiagonalization. SIAM J. Matrix Anal. Appl., Volume 29, No 3, pp. 927-953.

(4) **Stewart, G.W., 2007:** Block Gram-Schmidt Orthogonalization. Report TR-4823, Department of Computer Science, College Park, University of Maryland.

6.19.7 `subroutine bd_cmp3 ( mat, d, e, gen_p, failure )`

Purpose

BD_CMP3 reduces a m-by-n matrix MAT with m >= n to upper bidiagonal form BD by an orthogonal transformation :

Q' * MAT * P = BD

where Q and P are orthogonal.

BD_CMP3 computes BD and P using the one-sided Ralha-Barlow bidiagonal reduction algorithm.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the general m-by-n matrix to be reduced.

On exit, P is stored in MAT(1:n,1:n). See Further Details.

The shape of MAT must verify: size(MAT, 1) >= size(MAT, 2) = n .

D (OUTPUT) real(stnd), dimension(:) The diagonal elements of the bidiagonal matrix BD

The size of D must be size(MAT, 2) = n .

E (OUTPUT) real(stnd), dimension(:) The off-diagonal elements of the bidiagonal matrix BD:

E(i) = BD(i-1,i) for i = 2,3,. . . ,n;

The size of E must be size(MAT, 2) = n .

GEN_P (INPUT) logical(lgl) If:

- GEN_P = true : the orthogonal matrix P is generated in MAT(1:n,1:n) on output of the subroutine.
- GEN_P = false : the orthogonal matrix is stored in factored form as products of elementary reflectors in the lower triangle of the array MAT(1:n,1:n).

See further details.

FAILURE (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE = false : indicates successful exit ;
- FAILURE = true : indicates that MAT is nearly singular.

Further Details

This subroutine is an implementation of the Ralha-Barlow one-sided method to reduce a rectangular matrix MAT to bidiagonal form BD. Q is computed by a recurrence relationship (but is not stored) and P as a product of n-1 elementary reflectors (e.g., Householder transformations):

P = G(1) * G(2) * . . . * G(n-1)

Each G(i) has the form:

G(i) = I + taup * v * v'

where taup is a real scalar, and v is a real vector. IF GEN_P is set to false, the n-1 G(i) elementary reflectors are stored in the lower triangle of the array MAT. For the G(i) reflector, taup is stored in MAT(i+1,1) and v is stored in MAT(i+1:n,i+1). IF GEN_P is set to true, P is generated in MAT(:n,:n).

In addition, MAT(1,1) is set to -1 if GEN_P=false and is equal to 1 if GEN_P=true. In other words, the value of MAT(1,1) indicates if the orthogonal matrix P is stored in factored form or not in MAT. Note that

if n is equal to 1, no elementary reflectors are needed and consequently MAT(1,1) (e.g., P(1,1)) is set to 1, independently of the value of GEN_P.

This is the blocked version of the algorithm. See the references (1), (2) and (3) for further details. Note also that the blocked algorithm implemented here is more efficient than the version described in the reference (3). Furthermore the algorithm is parallelized if OPENMP is used.

The reference (2) also explains how to handle the case of an exactly singular matrix MAT (a very rare event). However, this case is not implemented here as this subroutine outputs only BD and P.

For further details, see:

(1) **Ralha, R.M.S., 2003:** One-sided reduction to bidiagonal form. Linear Algebra Appl., No 358, pp. 219-238.

(2) **Barlow, J.L., Bosner, N., and Drmac, Z., 2005:** A new stable bidiagonal reduction algorithm. Linear Algebra Appl., No 397, pp. 35-84.

(3) **Bosner, N., and Barlow, J.L., 2007:** Block and Parallel versions of one-sided bidiagonalization. SIAM J. Matrix Anal. Appl., Volume 29, No 3, pp. 927-953.

6.19.8 `subroutine ortho_gen_bd ( mat, tauq, taup, p )`

Purpose

ORTHO_GEN_BD generates the real orthogonal matrices Q and P determined by BD_CMP when reducing a m-by-n real matrix MAT to bidiagonal form:

MAT = Q * BD * P'.

Q and P are defined as products of elementary reflectors H(i) and G(i), respectively, determined by BD_CMP and stored in its array arguments MAT, TAUQ and TAUP.

If m >= n:

- Q = H(1) * H(2) * ... * H(n) and ORTHO_GEN_BD returns the first n columns of Q in MAT;
- P = G(1) * G(2) * ... * G(n-1) and ORTHO_GEN_BD returns P as an n-by-n matrix in P.

If m < n:

- Q = H(1) * H(2) * ... * H(m-1) and ORTHO_GEN_BD returns Q as an m-by-m matrix in MAT(1:m,1:m);
- P = G(1) * G(2) * ... * G(m) and ORTHO_GEN_BD returns the first m columns of P, in P.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the vectors which define the elementary reflectors H(i) and G(i), as returned by BD_CMP in its array argument MAT.

On exit, the first min(m,n) columns of Q are stored in MAT(1:m,1:min(m,n)).

TAUQ (INPUT) real(stnd), dimension(:) TAUQ(i) must contain the scalar factor of the elementary reflector H(i), which determines Q, as returned by BD_CMP in its array argument TAUQ.

The size of TAUQ must verify: size(TAUQ) = min(m,n) .

TAUP (INPUT) real(stnd), dimension(:) TAUP(i) must contain the scalar factor of the elementary reflector G(i), which determines P, as returned by BD_CMP in its array argument TAUP.

The size of TAUP must verify: size(TAUP) = min(m,n) .

P (OUTPUT) real(stnd), dimension(:,:) On exit, the first min(m,n) columns of the n-by-n matrix P

The shape of p must verify:

- size(P, 1) = n ,
- size(P, 2) = min(m,n).

Further Details

This subroutine used a blocked algorithm for agregating the Householder transformations (e.g. the elementary reflectors) stored in MAT and generating the orthogonal matrices Q and P of the bidiagonal decomposition of MAT.

Furthermore, the computations are parallelized if OPENMP is used.

For further details on the bidiagonal reduction algorithm and its use or the blocked algorithm used here, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(4) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.19.9 `subroutine ortho_gen_bd2 ( mat, tauq, taup, q_pt )`

Purpose

ORTHO_GEN_BD2 generates the real orthogonal matrices Q and P' determined by BD_CMP when reducing a m-by-n real matrix MAT to bidiagonal form:

MAT = Q * BD * P'

Q and P' are defined as products of elementary reflectors H(i) and G(i), respectively, determined by BD_CMP and stored in its array arguments MAT, TAUQ and TAUP.

If m >= n:

- Q = H(1) * H(2) * ... * H(n) and ORTHO_GEN_BD2 returns the first n columns of Q in MAT;
- P' = G(n-1) * ... * G(2) * G(1) and ORTHO_GEN_BD2 returns P' as an n-by-n matrix in Q_PT.

If m < n:

- Q = H(1) * H(2) * ... * H(m-1) and ORTHO_GEN_BD2 returns Q as an m-by-m matrix in Q_PT;
- P' = G(m) * ... * G(2) * G(1) and ORTHO_GEN_BD2 returns the first m rows of P', in MAT.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the vectors which define the elementary reflectors H(i) and G(i), as returned by BD_CMP in its array argument MAT.

On exit:

- the first n columns of Q if m >= n ;
- the first m rows of P' if m < n .

TAUQ (INPUT) real(stnd), dimension(:) TAUQ(i) must contain the scalar factor of the elementary reflector H(i), which determines Q, as returned by BD_CMP in its array argument TAUQ.

The size of TAUQ must verify: size(TAUQ) = min(m,n) .

TAUP (INPUT) real(stnd), dimension(:) TAUP(i) must contain the scalar factor of the elementary reflector G(i), which determines P', as returned by BD_CMP in its array argument TAUP.

The size of TAUP must verify: size(TAUP) = min(m,n) .

Q_PT (OUTPUT) real(stnd), dimension(:,:) On exit:

- the n-by-n matrix P' if m >= n ;
- the m-by-m matrix Q if m < n .

The shape of Q_PT must verify: size(Q_PT, 1) = size(Q_PT, 2) = min(m,n).

Further Details

This subroutine used a blocked algorithm for agregating the Householder transformations (e.g. the elementary reflectors) stored in MAT and generating the orthogonal matrices Q and P of the bidiagonal decomposition of MAT.

Furthermore, the computations are parallelized if OPENMP is used.

For further details on the bidiagonal reduction algorithm and its use or the blocked algorithm, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(4) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.19.10 `subroutine ortho_gen_q_bd ( mat, tauq )`

Purpose

ORTHO_GEN_Q_BD generates the real orthogonal matrix Q determined by BD_CMP when reducing a m-by-n real matrix MAT to bidiagonal form:

MAT = Q * BD * P'

Q is defined as products of elementary reflectors H(i) determined by BD_CMP and stored in its array arguments MAT and TAUQ.

If m >= n:

- Q = H(1) * H(2) * ... * H(n) and ORTHO_GEN_Q_BD returns the first n columns of Q in MAT.

If m < n:

- Q = H(1) * H(2) * ... * H(m-1) and ORTHO_GEN_Q_BD returns Q as an m-by-m matrix in MAT(:m,:m).

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the vectors which define the elementary reflectors H(i), as returned by BD_CMP.

On exit, the first min(m,n) columns of Q.

TAUQ (INPUT) real(stnd), dimension(:) TAUQ(i) must contain the scalar factor of the elementary reflector H(i), which determines Q, as returned by BD_CMP in its array argument TAUQ.

The size of TAUQ must verify: size(TAUQ) = min(m,n) .

Further Details

This subroutine used a blocked algorithm for agregating the Householder transformations (e.g. the elementary reflectors) stored in MAT and generating the orthogonal matrix Q of the bidiagonal decomposition of MAT.

Furthermore, the computations are parallelized if OPENMP is used.

For further details on the bidiagonal reduction algorithm and its use or the blocked algorithm, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(4) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.19.11 `subroutine ortho_gen_p_bd ( mat, taup, p )`

Purpose

ORTHO_GEN_P_BD generates the real orthogonal matrix P determined by BD_CMP when reducing a m-by-n real matrix MAT to bidiagonal form:

MAT = Q * BD * P'

P is defined as products of elementary reflectors G(i) determined by BD_CMP and stored in its array arguments MAT and TAUP.

If m >= n:

- P = G(1) * G(2) * ... * G(n-1) and ORTHO_GEN_P_BD returns P as an n-by-n matrix in P.

If m < n:

- P = G(1) * G(2) * ... * G(m) and ORTHO_GEN_P_BD returns the first m columns of P, in P.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the vectors which define the elementary reflectors G(i), as returned by BD_CMP in its array argument MAT.

TAUP (INPUT) real(stnd), dimension(:) TAUP(i) must contain the scalar factor of the elementary reflector G(i), which determines P, as returned by BD_CMP in its array argument TAUP.

The size of TAUP must verify: size(TAUP) = min(m,n) .

P (OUTPUT) real(stnd), dimension(:,:) On exit, the first min(m,n) columns of the n-by-n matrix P

The shape of p must verify:

- size(P, 1) = n ,
- size(P, 2) = min(m,n).

Further Details

This subroutine used a blocked algorithm for agregating the Householder transformations (e.g. the elementary reflectors) stored in MAT and generating the orthogonal matrix P of the bidiagonal decomposition of MAT.

Furthermore, the computations are parallelized if OPENMP is used.

For further details on the bidiagonal reduction algorithm and its use or the blocked algorithm, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(4) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.19.12 `subroutine apply_q_bd ( mat, tauq, c, left, trans )`

Purpose

APPLY_Q_BD overwrites the general real m-by-n matrix C with:

- Q * C if LEFT = true and TRANS = false ;
- Q' * C if LEFT = true and TRANS = true ;
- C * Q if LEFT = false and TRANS = false ;
- C * Q' if LEFT = false and TRANS = true .

Here Q is the orthogonal matrix determined by BD_CMP when reducing a real matrix MAT to bidiagonal form:

MAT = Q * BD * P'

and Q is defined as products of elementary reflectors H(i).

Let nq = m if LEFT = true and nq = n if LEFT = false. Thus nq is the order of the orthogonal matrix Q that is applied. MAT is assumed to have been an nq-by-k matrix and

Q = H(1) * H(2) * . . . * H(k) , if nq >= k ;

or

Q = H(1) * H(2) * . . . * H(nq-1) , if nq < k .

Arguments

MAT (INPUT) real(stnd), dimension(:,:) The vectors which define the elementary reflectors H(i), whose products determine the matrix Q, as returned by BD_CMP. MAT must be specified as in BD_CMP and is not modified by the routine.

The shape of MAT must verify:

- if LEFT = true : size(C, 1) = size(MAT, 1) = nq ;
- if LEFT = false : size(C, 2) = size(MAT, 1) = nq .

TAUQ (INPUT) real(stnd), dimension(:) TAUQ(i) must contain the scalar factor of the elementary reflector H(i) which determines Q, as returned by BD_CMP in the array argument TAUQ.

The size of TAUQ must verify: size(TAUQ) = min(size(MAT,1) , size(MAT,2)) .

C (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m by n matrix C.

On exit, C is overwritten by Q * C or Q' * C or C * Q' or C * Q .

The shape of C must verify:

- if LEFT = true : size(C, 1) = size(MAT, 1) = nq ;
- if LEFT = false : size(C, 2) = size(MAT, 1) = nq .

LEFT (INPUT) logical(lgl) On entry, if:

- LEFT= true : apply Q or Q' from the left
- LEFT= false : apply Q or Q' from the right

TRANS (INPUT) logical(lgl) On entry, if:

- TRANS = false : apply Q (no transpose)
- TRANS = true : apply Q' (transpose)

Further Details

This subroutine is adapted from the routine DORMBR in LAPACK.

This subroutine used a blocked algorithm for agregating the Householder transformations (e.g. the elementary reflectors) stored in the lower triangle of MAT and applying the orthogonal matrix Q of the bidiagonal factorization to the real m-by-n matrix C.

Furthermore, the computations are parallelized if OPENMP is used.

For further details on the bidiagonal reduction algorithm and the blocked version of the algorithm used here, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(4) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.19.13 `subroutine apply_p_bd ( mat, taup, c, left, trans )`

Purpose

APPLY_P_BD overwrites the general real m-by-n matrix C with

- P * C if LEFT = true and TRANS = false ;
- P' * C if LEFT = true and TRANS = true ;
- C * P if LEFT = false and TRANS = false ;
- C * P' if LEFT = false and TRANS = true .

Here P is the orthogonal matrix determined by BD_CMP when reducing a real matrix MAT to bidiagonal form:

MAT = Q * BD * P'

and P is defined as products of elementary reflectors G(i).

Let np = m if LEFT = true and np = n if LEFT = false. Thus np is the order of the orthogonal matrix P that is applied. MAT is assumed to have been an k-by-np matrix and

P = G(1) * G(2) * ... * G(k) , if k < np ;

or

P = G(1) * G(2) * ... * G(np-1) , if k >= np .

Arguments

MAT (INPUT) real(stnd), dimension(:,:) The vectors which define the elementary reflectors G(i), whose products determine the matrix P, as returned by BD_CMP. MAT must be specified as in BD_CMP and is not modified by the routine.

The shape of MAT must verify:

- if LEFT = true : size(C, 1) = size(MAT, 2) = np ;
- if LEFT = false : size(C, 2) = size(MAT, 2) = np .

TAUP (INPUT) real(stnd), dimension(:) TAUP(i) must contain the scalar factor of the elementary reflector G(i) which determines P, as returned by BD_CMP in the array argument TAUP.

The size of TAUP must verify: size(TAUP) = min(size(MAT,1) , size(MAT,2)) .

C (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m by n matrix C.

On exit, C is overwritten by P * C or P' * C or C * P or C * P'.

The shape of C must verify:

- if LEFT = true : size(C, 1) = size(MAT, 2) = np ;
- if LEFT = false : size(C, 2) = size(MAT, 2) = np .

LEFT (INPUT) logical(lgl) On entry, if:

- LEFT= true : apply P or P' from the left
- LEFT= false : apply P or P' from the right

TRANS (INPUT) logical(lgl) On entry, if:

- TRANS = false : apply P (no transpose)
- TRANS = true : apply P' (transpose)

Further Details

This subroutine is adapted from the routine DORMBR in LAPACK.

This subroutine used a blocked algorithm for agregating the Householder transformations (e.g. the elementary reflectors) stored in the upper triangle of MAT and applying the orthogonal matrix P of the bidiagonal factorization to the real m-by-n matrix C.

Furthermore, the computations are parallelized if OPENMP is used.

For further details on the bidiagonal reduction algorithm and the blocked version of the algorithm used here, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(3) **Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., 1989:** Block reduction of matrices to condensed form for eigenvalue computations. J. of Computational and Applied Mathematics, Vol. 27, pp. 215-227.

(4) **Walker, H.F., 1988:** Implementation of the GMRES method using Householder transformations. Siam J. Sci. Stat. Comput., Vol. 9, No 1, pp. 152-163.

6.19.14 subroutine bd_svd (upper, d, e, failure, u, v, sort, maxiter, max_francis_steps, perfect_shift, bisect, dqds)

Purpose

BD_SVD computes the singular value decomposition (SVD) of a real n-by-n (upper or lower) bidiagonal matrix B:

B = Q * S * P'

, where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and, Q and P are orthogonal matrices (P' denotes the transpose of P).

The routine computes S, U * Q, and V * P, for given real input matrices U, V.

Arguments

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : B is upper bidiagonal ;
- UPPER = false : B is lower bidiagonal.

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix B.

On exit, D contains the singular values of B.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the bidiagonal matrix whose SVD is desired. E(1) is arbitrary.

On exit, E is destroyed.

The size of E must verify: size(E) = size(D) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit ;
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the bidiagonal SVD of B.

U (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the matrix U.

On exit, U is overwritten by U * Q.

The shape of U must verify:

- size(U, 1) > 0 ;
- size(U, 2) = size(D) = n .

V (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the matrix V.

On exit, V is overwritten by V * P.

The shape of V must verify:

- size(V, 1) > 0 ;
- size(V, 2) = size(D) = n .

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The singular vectors are rearranged accordingly.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the algorithm. The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * n. Convergence usually occurs in about 2 * n QR sweeps.

The default is 10.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the implicit QR algorithm. MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of n and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm applied to the associated n-by-n symmetric tridiagonal matrix B * B' whose eigenvalues are the squares of the singular values of B.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm applied to the associated n-by-n symmetric tridiagonal matrix B * B' whose eigenvalues are the squares of the singular values of B.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

If, on entry, arguments U and V are n-by-n identity matrices, on exit they are replaced by Q and P, respectively.

This subroutine is adapted from subroutine QRBD given in the reference (1), with modifications suggested in the references (2) and (3) for the application of a set of Givens rotations to the singular vectors, and extensions to the bidiagonal case of the perfect shift strategy presented in the references (4) and (5) for the tridiagonal case.

Furthermore, the computation of the singular vectors is parallelized if OPENMP is used.

Note, finally, that the bidiagonal matrix is not scaled before computing the singular values and vectors. If some of the elements of the bidiagonal matrix are very small or large, it may be appropriate to scale the bidiagonal matrix before calling BD_SVD.

For further details, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Lang, B., 1998:** Using level 3 BLAS in rotation-based algorithms. Siam J. Sci. Comput., Vol. 19, 626-634.

(3) **Van Zee, F.G., Van de Geijn, R., and Quintana-Orti, G., 2011:** Restructuring the QR Algorithm for High-Performance Application of Givens Rotations. FLAME Working Note 60. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-11-36.

(4) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, 1013-1034.

(5) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

6.19.15 subroutine bd_svd (upper, d, e, failure, u, sort, maxiter, max_francis_steps, perfect_shift, bisect, dqds)

Purpose

BD_SVD computes the singular value decomposition (SVD) of a real n-by-n (upper or lower) bidiagonal matrix B:

B = Q * S * P'

, where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and, Q and P are orthogonal matrices (P' denotes the transpose of P).

The routine computes S and U * Q for a given real input matrix U.

Arguments

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : B is upper bidiagonal ;
- UPPER = false : B is lower bidiagonal.

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix B.

On exit, D contains the singular values of B.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the bidiagonal matrix whose SVD is desired. E(1) is arbitrary.

On exit, E is destroyed.

The size of E must verify: size(E) = size(D) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit ;
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the bidiagonal SVD of B.

U (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the matrix U.

On exit, U is overwritten by U * Q.

The shape of U must verify:

- size(U, 1) > 0 ;
- size(U, 2) = size(D) = n .

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The singular vectors U are rearranged accordingly.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the algorithm. The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * n. Convergence usually occurs in about 2 * n QR sweeps.

The default is 10.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the implicit QR algorithm. MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of n and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm applied to the associated n-by-n symmetric tridiagonal matrix B * B' whose eigenvalues are the squares of the singular values of B.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm applied to the associated n-by-n symmetric tridiagonal matrix B * B' whose eigenvalues are the squares of the singular values of B.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

If argument U is a n-by-n identity matrix, on exit it is replaced by Q.

This subroutine is adapted from subroutine QRBD given in the reference (1), with modifications suggested in the references (2) and (3) for the application of a set of Givens rotations to the singular vectors, and

extensions to the bidiagonal case of the perfect shift strategy presented in the references (4) and (5) for the tridiagonal case.

Furthermore, the computation of the singular vectors is parallelized if OPENMP is used.

Note, finally, that the bidiagonal matrix is not scaled before computing the singular values and vectors. If some of the elements of the bidiagonal matrix are very small or large, it may be appropriate to scale the bidiagonal matrix before calling BD_SVD.

For further details, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Lang, B., 1998:** Using level 3 BLAS in rotation-based algorithms. Siam J. Sci. Comput., Vol. 19, 626-634.

(3) **Van Zee, F.G., Van de Geijn, R., and Quintana-Orti, G., 2011:** Restructuring the QR Algorithm for High-Performance Application of Givens Rotations. FLAME Working Note 60. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-11-36.

(4) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, 1013-1034.

(5) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

6.19.16 `subroutine bd_svd ( upper, d, e, failure, sort, maxiter )`

Purpose

BD_SVD computes the singular values, S, of a real n-by-n (upper or lower) bidiagonal matrix B:

B = Q * S * P'

, where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and, Q and P are orthogonal matrices (P' denotes the transpose of P).

The singular values are computed by the bidiagonal implicit QR method. See the references (1) and (2) for details.

Arguments

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : B is upper bidiagonal ;
- UPPER = false : B is lower bidiagonal.

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix B.

On exit, D contains the singular values of B.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the bidiagonal matrix whose singular values are desired. E(1) is arbitrary.

On exit, E is destroyed.

The size of E must verify: size(E) = size(D) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit ;
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the bidiagonal SVD of B.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the algorithm. The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * n. Convergence usually occurs in about 2 * n QR sweeps.

The default is 10.

Further Details

This subroutine is adapted from subroutine QRBD in the reference (1).

For further details, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.19.17 subroutine bd_svd2 (upper, d, e, failure, u, vt, sort, maxiter, max_francis_steps, perfect_shift, bisect, dqds)

Purpose

BD_SVD2 computes the singular value decomposition (SVD) of a real n-by-n (upper or lower) bidiagonal matrix B:

B = Q * S * P'

, where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and, Q and P are orthogonal matrices (P' denotes the transpose of P).

The routine computes S, U * Q, and P' * VT, for given real input matrices U, VT.

Arguments

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : B is upper bidiagonal ;
- UPPER = false : B is lower bidiagonal.

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix B.

On exit, D contains the singular values of B.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the bidiagonal matrix whose SVD is desired. E(1) is arbitrary.

On exit, E is destroyed.

The size of E must verify: size(E) = size(D) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit ;
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the bidiagonal SVD of B.

U (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the matrix U.

On exit, U is overwritten by U * Q.

The shape of U must verify:

- size(U, 1) > 0 ;
- size(U, 2) = size(D) = n .

VT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the matrix VT.

On exit, VT is overwritten by P' * VT.

The shape of VT must verify:

- size(VT, 1) = size(D) = n ;
- size(VT, 2) > 0 .

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The singular vectors are rearranged accordingly.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the algorithm. The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * n. Convergence usually occurs in about 2 * n QR sweeps.

The default is 10.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the implicit QR algorithm. MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of n and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm applied to the associated n-by-n symmetric tridiagonal matrix B * B' whose eigenvalues are the squares of the singular values of B.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm applied to the associated n-by-n symmetric tridiagonal matrix B * B' whose eigenvalues are the squares of the singular values of B.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

If arguments U and VT are n-by-n identity matrices, on exit they are replaced by Q and P', respectively.

This subroutine is adapted from subroutine QRBD given in the reference (1), with modifications suggested in the references (2) and (3) for the application of a set of Givens rotations to the singular vectors, and extensions to the bidiagonal case of the perfect shift strategy presented in the references (4) and (5) for the tridiagonal case.

Furthermore, the computation of the singular vectors is parallelized if OPENMP is used.

Note, finally, that the bidiagonal matrix is not scaled before computing the singular values and vectors. If some of the elements of the bidiagonal matrix are very small or large, it may be appropriate to scale the bidiagonal matrix before calling BD_SVD2.

For further details, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Lang, B., 1998:** Using level 3 BLAS in rotation-based algorithms. Siam J. Sci. Comput., Vol. 19, 626-634.

(3) **Van Zee, F.G., Van de Geijn, R., and Quintana-Orti, G., 2011:** Restructuring the QR Algorithm for High-Performance Application of Givens Rotations. FLAME Working Note 60. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-11-36.

(4) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, 1013-1034.

(5) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

6.19.18 subroutine bd_svd2 (upper, d, e, failure, u, sort, maxiter, max_francis_steps, perfect_shift, bisect, dqds)

Purpose

BD_SVD2 computes the singular value decomposition (SVD) of a real n-by-n (upper or lower) bidiagonal matrix B:

B = Q * S * P'

, where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and, Q and P are orthogonal matrices (P' denotes the transpose of P).

The routine computes S and U * Q for a given real input matrix U.

Arguments

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : B is upper bidiagonal ;
- UPPER = false : B is lower bidiagonal.

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix B.

On exit, D contains the singular values of B.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the bidiagonal matrix whose SVD is desired. E(1) is arbitrary.

On exit, E is destroyed.

The size of E must verify: size(E) = size(D) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit ;
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the bidiagonal SVD of B.

U (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the matrix U.

On exit, U is overwritten by U * Q.

The shape of U must verify:

- size(U, 1) > 0 ;
- size(U, 2) = size(D) = n .

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The singular vectors U are rearranged accordingly.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the algorithm. The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * n. Convergence usually occurs in about 2 * n QR sweeps.

The default is 10.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the implicit QR algorithm. MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of n and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm applied to the associated n-by-n symmetric tridiagonal matrix B * B' whose eigenvalues are the squares of the singular values of B.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm applied to the associated n-by-n symmetric tridiagonal matrix B * B' whose eigenvalues are the squares of the singular values of B.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

If argument U is a n-by-n identity matrix, on exit it is replaced by Q.

This subroutine is adapted from subroutine QRBD given in the reference (1), with modifications suggested in the references (2) and (3) for the application of a set of Givens rotations to the singular vectors, and extensions to the bidiagonal case of the perfect shift strategy presented in the references (4) and (5) for the tridiagonal case.

Furthermore, the computation of the singular vectors is parallelized if OPENMP is used.

Note, finally, that the bidiagonal matrix is not scaled before computing the singular values and vectors. If some of the elements of the bidiagonal matrix are very small or large, it may be appropriate to scale the bidiagonal matrix before calling BD_SVD2.

For further details, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Lang, B., 1998:** Using level 3 BLAS in rotation-based algorithms. Siam J. Sci. Comput., Vol. 19, 626-634.

(3) **Van Zee, F.G., Van de Geijn, R., and Quintana-Orti, G., 2011:** Restructuring the QR Algorithm for High-Performance Application of Givens Rotations. FLAME Working Note 60. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-11-36.

(4) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, 1013-1034.

(5) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

6.19.19 subroutine bd_svd2 (upper, d, e, failure, sort, maxiter)

Purpose

BD_SVD2 computes the singular values, S, of a real n-by-n (upper or lower) bidiagonal matrix B:

B = Q * S * P'

, where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and, Q and P are orthogonal matrices (P' denotes the transpose of P).

The singular values are computed by the bidiagonal implicit QR method. See the references (1) and (2) for details.

Arguments

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : B is upper bidiagonal ;
- UPPER = false : B is lower bidiagonal.

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix B.

On exit, D contains the singular values of B.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the bidiagonal matrix whose singular values are desired. E(1) is arbitrary.

On exit, E is destroyed.

The size of E must verify: size(E) = size(D) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit ;
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the bidiagonal SVD of B.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the algorithm. The algorithm fails to converge if the number of QR sweeps exceeds MAXITER * n. Convergence usually occurs in about 2 * n QR sweeps.

The default is 10.

Further Details

This subroutine is adapted from subroutine QRBD in the reference (1).

For further details, see:

(1) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(2) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.19.20 subroutine bd_singval (d, e, nsing, s, failure, sort, vector, abstol, ls, theta, scaling, init)

Purpose

BD_SINGVAL computes all or some of the greatest singular values of a real n-by-n (upper or lower) bidiagonal matrix B by a bisection algorithm.

The Singular Value Decomposition of B is:

B = Q * S * P'

where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and, Q and P are orthogonal matrices (P' denotes the transpose of P).

Arguments

D (INPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix B.

E (INPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the bidiagonal matrix whose singular values are desired. E(1) is arbitrary.

The size of E must verify: size(E) = size(D) .

NSING (OUTPUT) integer(i4b) On output, NSING specifies the number of singular values which have been computed. Note that NSING may be greater than the optional argument LS, if multiple singular values at index LS make unique selection impossible.

If none of the optional arguments LS and THETA are used, NSING is set to size(D) and all the singular values are computed.

S (OUTPUT) real(stnd), dimension(:) On exit, S(1:NSING) contains the NSING greatest singular values of B. The other values in S (S(NSING+1:size(D))) are flagged by a quiet NAN.

The size of S must verify: size(S) = size(D) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit and the bisection algorithm converged for all the computed singular values to the desired accuracy ;
- FAILURE = true : indicates that some or all of the singular values failed to converge or were not computed. This is generally caused by unexpectedly inaccurate arithmetic. The sign of the incorrect singular values is set to negative.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. For other values of SORT nothing is done and S(:nsing) may not be sorted in decreasing order of of magnitude.

VECTOR (INPUT, OPTIONAL) logical(lgl) On entry, if VECTOR is set to TRUE, a vectorized version of the bisection algorithm is used.

The default is VECTOR=false.

ABSTOL (INPUT, OPTIONAL) real(stnd) On entry, the absolute tolerance for the singular values. A singular value (or cluster) is considered to be located if it has been determined to lie in an interval whose width is ABSTOL or less.

If ABSTOL is less than or equal to zero, or is not specified, then ULP * | T(GK) | will be used, where | T(GK) | means the 1-norm of the GOLUB-KAHAN tridiagonal form of the bidiagonal matrix B and ULP is the machine precision (distance from 1 to the next larger floating point number).

Singular values will be computed most accurately when ABSTOL is set to the square root of the underflow threshold, sqrt(LAMCH('S')), not zero.

LS (INPUT, OPTIONAL) integer(i4b) On entry, LS specifies the number of singular values which must be computed by the subroutine. On output, NSING may be different than LS if multiple singular values at index LS make unique selection impossible.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

LS must be greater than 0 and less or equal to size(D) .

The default is LS = size(D).

THETA (INPUT, OPTIONAL) real(stnd) On entry, THETA specifies that the singular values which are greater or equal to THETA must be computed. If none of the singular values are greater or equal to THETA, NSING is set to zero and S(:) to a quiet NAN.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

The default is THETA = 0.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the bidiagonal matrix B is scaled before computing the singular values.

The default is to scale the bidiagonal matrix.

INIT (INPUT, OPTIONAL) logical(lgl) On entry, if INIT=true the initial intervals for the bisection steps are computed from estimates of the eigenvalues of the associated B' * B tridiagonal matrix obtained from the Pal-Walker-Kahan algorithm.

The default is not to use the Pal-Walker-Kahan algorithm.

Further Details

Let S(i), i=1,...,N=size(D), be the N singular values of the bidiagonal matrix B in decreasing order of magnitude. BD_SINGVAL then computes the LS largest singular values (or the singular values which are greater or equal to THETA) of B by a bisection method (see the reference (1) below, Sec.8.5). The bisection method is applied to an associated 2N by 2N symmetric tridiagonal matrix T (the so-called GOLUB-KAHAN form of B) whose eigenvalues are the singular values of B and their negatives (see the reference (2) below, Sec.3.3).

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

6.19.21 subroutine bd_singval2 (d, e, nsing, s, failure, sort, vector, abstol, ls, theta, scaling, init)

Purpose

BD_SINGVAL2 computes all or some of the greatest singular values of a real n-by-n (upper or lower) bidiagonal matrix B by a bisection algorithm.

The Singular Value Decomposition of B is:

B = Q * S * P'

where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and, Q and P are orthogonal matrices (P' denotes the transpose of P).

Arguments

D (INPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix B.

E (INPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the bidiagonal matrix whose singular values are desired. E(1) is arbitrary.

The size of E must verify: size(E) = size(D) .

NSING (OUTPUT) integer(i4b) On output, NSING specifies the number of singular values which have been computed. Note that NSING may be greater than the optional argument LS, if multiple singular values at index LS make unique selection impossible.

If none of the optional arguments LS and THETA are used, NSING is set to size(D) and all the singular values are computed.

S (OUTPUT) real(stnd), dimension(:) On exit, S(1:NSING) contains the NSING greatest singular values of B. The other values in S (S(NSING+1:size(D))) are flagged by a quiet NAN.

The size of S must verify: size(S) = size(D) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit and the bisection algorithm converged for all the computed singular values to the desired accuracy ;
- FAILURE = true : indicates that some or all of the singular values failed to converge or were not computed. This is generally caused by unexpectedly inaccurate arithmetic. The sign of the incorrect singular values is set to negative.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. For other values of SORT nothing is done and S(:nsing) may not be sorted in decreasing order of of magnitude.

VECTOR (INPUT, OPTIONAL) logical(lgl) On entry, if VECTOR is set to TRUE, a vectorized version of the bisection algorithm is used.

The default is VECTOR=false.

ABSTOL (INPUT, OPTIONAL) real(stnd) On entry, the absolute tolerance for the singular values. A singular value (or cluster) is considered to be located if its square has been determined to lie in an interval whose width is ABSTOL or less.

If ABSTOL is less than or equal to zero, or is not specified, then ULP * | B' * B | will be used, where | B' * B | means the 1-norm of the tridiagonal matrix B' * B (B' means the transpose of B) and ULP is the machine precision (distance from 1 to the next larger floating point number).

Singular values will be computed most accurately when ABSTOL is set to the square root of the underflow threshold, sqrt(LAMCH('S')), not zero.

LS (INPUT, OPTIONAL) integer(i4b) On entry, LS specifies the number of singular values which must be computed by the subroutine. On output, NSING may be different than LS if multiple singular values at index LS make unique selection impossible.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

LS must be greater than 0 and less or equal to size(D) .

The default is LS = size(D).

THETA (INPUT, OPTIONAL) real(stnd) On entry, THETA specifies that the singular values which are greater or equal to THETA must be computed. If none of the singular values are greater or equal to THETA, NSING is set to zero and S(:) to a quiet NAN.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

The default is THETA = 0.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the bidiagonal matrix B is scaled before computing the singular values.

The default is to scale the bidiagonal matrix.

INIT (INPUT, OPTIONAL) logical(lgl) On entry, if INIT=true the initial intervals for the bisection steps are computed from estimates of the eigenvalues of the associated B' * B tridiagonal matrix obtained from the Pal-Walker-Kahan algorithm.

The default is not to use the Pal-Walker-Kahan algorithm.

Further Details

Let S(i), i=1,. . . ,N=size(D), be the N singular values of the bidiagonal matrix B in decreasing order of magnitude. BD_SINGVAL2 then computes the LS largest singular values (or the singular values which are greater or equal to THETA) of B by a bisection method (see the reference (1) below, Sec.8.5). The bisection method is applied (implicitly) to the associated N by N symmetric tridiagonal matrix B' * B whose eigenvalues are the squares of the singular values of B by using the differential stationary form of the qd algorithm of Rutishauser (see the reference (2) below, Sec.3.1).

BD_SINGVAL2 is faster than BD_SINGVAL, however if relative accuracy for small singular values is required, BD_SINGVAL (which is based on the Golub-Kahan form of the bidiagonal matrix) is the best choice.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

6.19.22 subroutine bd_max_singval (d, e, nsing, s, failure, abstol, scaling)

Purpose

BD_MAX_SINGVAL computes the greatest singular value of a real n-by-n (upper or lower) bidiagonal matrix B by a bisection algorithm.

The Singular Value Decomposition of a bidiagonal matrix B is:

B = Q * S * P'

where S is a diagonal matrix with non-negative diagonal elements (the singular values of B), and, Q and P are orthogonal matrices (P' denotes the transpose of P).

Arguments

D (INPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix B.

E (INPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the bidiagonal matrix whose singular values are desired. E(1) is arbitrary.

The size of E must verify: size(E) = size(D) .

NSING (OUTPUT) integer(i4b) On output, NSING specifies the number of singular values which have been computed. Note that NSING may be greater than 1 if multiple singular values make unique selection of the greatest singular value impossible.

S (OUTPUT) real(stnd), dimension(:) On exit, S(1:NSING) contains the first NSING greatest singular values of B. The other values in S (S(NSING+1:size(D))) are flagged by a quiet NAN.

The size of S must verify: size(S) = size(D) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit and the bisection algorithm converged for all the computed singular values to the desired accuracy ;
- FAILURE = true : indicates that some or all of the singular values failed to converge or were not computed. This is generally caused by unexpectedly inaccurate arithmetic. The sign of the incorrect singular values is set to negative.

ABSTOL (INPUT, OPTIONAL) real(stnd) On entry, the absolute tolerance for the singular values. A singular value (or cluster) is considered to be located if its square has been determined to lie in an interval whose width is ABSTOL or less.

If ABSTOL is less than or equal to zero, or is not specified, then ULP * | B' * B | will be used, where | B' * B | means the 1-norm of the tridiagonal matrix B' * B (B' means the transpose of B) and ULP is the machine precision (distance from 1 to the next larger floating point number).

Singular values will be computed most accurately when ABSTOL is set to the square root of the underflow threshold, sqrt(LAMCH('S')), not zero.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the bidiagonal matrix B is scaled before computing the greatest singular values.

The default is to scale the bidiagonal matrix.

Further Details

BD_MAX_SINGVAL computes the largest singular value of B by a bisection method (see the reference (1) below, Sec.8.5).

The bisection method is applied (implicitly) to the associated n-by-n symmetric tridiagonal matrix B' * B whose eigenvalues are the squares of the singular values of B by using the differential stationary form of the qd algorithm of Rutishauser (see the reference (2) below, Sec.3.1).

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

6.19.23 `subroutine bd_lasq1 ( d, e, failure, maxiter, sort, scaling, ieee, aggdef2, max_win, freq, info )`

Purpose

BD_LASQ1 computes the singular values of a real n-by-n bidiagonal matrix BD (with diagonal D and off-diagonal E) by the differential quotient difference with shifts (dqds) algorithm and, optionally, with an early aggressive deflation strategy.

The singular values are computed to high relative accuracy, in the absence of denormalization, underflow and overflow. The algorithm was first presented in the reference (1) and the present implementation is described in the references (2), (3) and (4). The early aggressive deflation strategy is described in the reference (4).

Arguments

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, D contain the diagonal elements of the bidiagonal matrix BD whose SVD is desired.

On normal exit, D contains the singular values of BD (e.g., if FAILURE is false). On abnormal exit (e.g. if FAILURE is true), D is overwritten only if INFO is equal to 2.

See description of FAILURE and INFO arguments for further details.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, E(2:) contain the off-diagonal elements of the bidiagonal matrix BD whose singular values are desired. E(1) is arbitrary.

On normal exit, E is unchanged, but, on abnormal exit, E(2:) is overwritten if INFO is equal to 2.

See description of FAILURE and INFO arguments for further details.

The size of E must verify: size(E) = size(D) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit;
- FAILURE = true : indicates that the algorithm did not converge. The specific reasons for the failure are given by the optional argument INFO described below if this argument is specified in the call to BD_LASQ1.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of iterations in the inner loop of the dqds algorithm (see BD_LASQ2 and BD_LASQ2_AGGDEF2 subroutines). MAXITER must be greater than 10, otherwise the default value is used.

The default is 30.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true, the bidiagonal matrix BD is scaled before computing the singular values.

The default is to scale the bidiagonal matrix.

IEEE (INPUT, OPTIONAL) logical(lgl) On entry, logical flag for IEEE or non IEEE arithmetic. When IEEE is set to true, the division in the dqds loop is not protected from incurring an exception (division by zero or overflow) since the powerful feature of arithmetic units conforming to IEEE floating point standard 754 is that computation is not held up by an exception. At the end of the loop, the code tests whether an infinite or an NaN (not a number) occurred and acts appropriately.

The default is to test inside the subroutine if IEEE arithmetic is supported or not.

AGGDEF2 (INPUT, OPTIONAL) logical(lgl) On entry, if AGGDEF2=true, early aggressive deflation is used in conjunction with the dqds algorithm.

The default is to use early aggressive deflation only if the dimension of the input bidiagonal matrix is sufficiently large (e.g., if size(D)>=20000).

MAX_WIN (INPUT, OPTIONAL) integer(i4b) On entry, the maximum window size to be used in early aggressive deflation. MAX_WIN must be greater or equal to 5.

This optional argument has an effect only if AGGDEF2 is equal to true.

The default value is max(int(sqrt(real(size(D)))), 5).

FREQ (INPUT, OPTIONAL) integer(i4b) On entry, the frequency with which early aggressive deflation will be invoked. FREQ must be greater or equal to 1.

This optional argument has an effect only if AGGDEF2 is equal to true.

The default value is 16.

INFO (OUTPUT, OPTIONAL) integer(i4b) On exit:

- INFO = 0 : indicates successful exit;
- INFO = 1 : indicates a failure because a split was marked by a positive value in E;
- INFO = 2 : indicates that the algorithm did not converge and that the maximum number of iterations in the inner loop of the dqds algorithm is exceeded; On exit, D and E represent a matrix with the same singular values which the calling subroutine could use to finish the computation, or even feed back into BD_LASQ1;
- INFO = 3 : indicates that the termination criterion of the outer while loop of the algorithm (in BD_LASQ2 or BD_LASQ2_AGGDEF2 subroutines) was not met (e.g., the program created more than n=size(D) unreduced blocks);
- INFO = 4 : indicates that the maximum number of iterations in the inner loop of the dqds algorithm is exceeded (as if INFO=2), but that a bidiagonal matrix with the same singular values as the original one cannot be recovered.

Further Details

This subroutine is adapted from the DLASQ1 subroutine in LAPACK version 3.12.0 and the codes (e.g., DLASQDD1 subroutine) for performing early aggressive deflation given in the reference (4).

This subroutine is a variant of the dqds algorithm described in the references (2), (3) and (4). Early aggressive deflation strategy, as described in the reference (4), is included in this version of the dqds algorithm when the optional logical argument AGGDEF2 is used with the value true.

For further details on the dqds algorithm, see:

(1) **Fernando, K.V., and Parlett, B.N., 1994:** Accurate singular values and differential qd algorithms. Numer. Math., Volume 67, No 2, pp. 191-229.

(2) **Parlett, B.N., and Marques, O.A., 2000:** An implementation of the dqds algorithm (positive case). Linear Algebra Appl., Volume 309, pp. 217-259.

(3) **Li, S., Gu, M., and Parlett, B. N., 2014:** An Improved DQDS Algorithm. SIAM Journal on Scientific Computing, Volume 36, No 3, C290-C308.

(4) **Nakatsukasa, Y., Aishima, K., and Yamazaki, I., 2012:** dqds with aggressive early deflation. SIAM Journal on Matrix Analysis Applications, Volume 33, No 1, pp. 22-51.

6.19.24 `subroutine bd_dqds ( d, e, failure, maxiter, sort, scaling )`

Purpose

BD_DQDS computes all the singular values, S, of a real n-by-n (upper or lower) bidiagonal matrix BD:

BD = Q * S * P'

, where S is a diagonal matrix with non-negative diagonal elements (the singular values of BD), and, Q and P are orthogonal matrices (P' denotes the transpose of P).

The singular values are computed to high relative precision in the absence of denormalization, underflow and overflow with a variant of the differential quotient difference with shifts (dqds) algorithm described in the reference (3).

Arguments

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix BD.

On exit, D contains the singular values of BD if FAILURE is false, but is unchanged if FAILURE is true.

See description of FAILURE argument for further details.

E (INPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the bidiagonal matrix BD whose singular values are desired. E(1) is arbitrary.

On exit, E is unchanged.

The size of E must verify: size(E) = size(D) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit ;

- FAILURE = true : indicates that the algorithm did not converge and that the maximum number of iterations in the inner loop of the algorithm is exceeded. In that case, D is unchanged on exit.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of iterations in the inner loop of the dqds algorithm. MAXITER must be greater than 5, otherwise the default value is used.

The default is 30.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the bidiagonal matrix BD is scaled before computing the singular values.

The default is to scale the bidiagonal matrix.

Further Details

This subroutine is adapted from the variant of the dqds algorithm described in the reference (3), which uses a new shift strategy in the dqds algorithm.

BD_DQDS is a Fortran95/2003 translation and improved version of a Fortran77 subroutine available at:

- http://syskiso.fuee.u-fukui.ac.jp/~kkimur/LAPROGNC/LAPROGNC.html

For further details on the dqds algorithm and its implementation, see:

(1) **Fernando, K.V., and Parlett, B.N., 1994:** Accurate singular values and differential qd algorithms. Numer. Math., Volume 67, No 2, pp. 191-229.

(2) **Parlett, B.N., and Marques, O.A., 2000:** An implementation of the dqds algorithm (positive case). Linear Algebra Appl., Volume 309, pp. 217-259.

(3) **Yamashita, T., Kimura, K., Takata, M., and Nakamura, Y., 2013:** An application of the Kato-Temple inequality on matrix eigenvalues to the dqds algorithm for singular values. JSIAM Letters, Volume 5, pp. 21-24.

6.19.25 `subroutine las2 ( f, g, h, ssmin, ssmax )`

Purpose

LAS2 computes singular values of a 2-by-2 triangular matrix.

Arguments

F (INPUT) real(stnd) On entry, the (1,1) element of the 2-by-2 matrix.

G (INPUT) real(stnd) On entry, the (1,2) element of the 2-by-2 matrix.

H (INPUT) real(stnd) On entry, the (2,2) element of the 2-by-2 matrix.

SSMIN (OUTPUT) real(stnd) On exit, the smaller singular value.

SSMAX (OUTPUT) real(stnd) On exit, the larger singular value.

Further Details

This subroutine is adapted from the DLAS2 subroutine from LAPACK version 3.12.0.

Barring over/underflow, all output quantities are correct to within a few units in the last place (ulps), even in the absence of a guard digit in addition/subtraction.

In IEEE arithmetic, the code works also correctly if one matrix element is infinite.

6.19.26 `function singvalues ( mat, sort, mul_size, maxiter, dqds )`

Purpose

Function SINGVALUES computes the singular values of a real m-by-n matrix MAT. The Singular Value Decomposition (SVD) is written

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative.

The original matrix MAT is first reduced to upper or lower bidiagonal form BD by an orthogonal transformation:

Q' * MAT * P = BD

where Q and P are orthogonal. The singular values SIGMA of the bidiagonal matrix BD are then computed by the bidiagonal implicit QR algorithm (if DQDS=false) or the dqds algorithm (if DQDS=true). If these algorithms fail to converge, function SINGVALUES returns a min(m,n)-vector filled with a quiet NaN.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MUL_SIZE (INPUT, OPTIONAL) integer(i4b) Internal parameter. MUL_SIZE must verify: 1 <= MUL_SIZE <= max(m,n), otherwise a default value is used. For better performance, at the expense of more workspace, a large value can be used.

The default is 32.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm.

The bidiagonal SVD algorithm of an intermediate bidiagonal form BD of MAT fails to converge if the number of QR sweeps exceeds MAXITER*min(m,n). Convergence usually occurs in about 2*min(m,n) QR sweeps.

This argument has no effect if DQDS is equal to true.

The default is 10.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values of the intermediate min(m,n)-by-min(m,n) bidiagonal matrix BD are computed.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition. Moreover, the dqds algorithm is usually also faster than the bidiagonal implicit QR algorithm.

If DQDS is set to false, singular values are computed with the bidiagonal implicit QR algorithm applied to the associated min(m,n)-by-min(m,n) bidiagonal matrix BD.

The default is true.

Further Details

Computing the singular values of a rectangular matrix in function SINGVALUES consists of two steps:

1) reduction of the rectangular matrix to bidiagonal form BD, see the references (1) and (2);

2) computation of the singular values of the min(m,n)-by-min(m,n) bidiagonal matrix BD by the bidiagonal implicit QR or dqds algorithms, see the references(1), (2), (3) and (4).

Note that if max(m,n) is much larger than min(m,n) the rectangular matrix is first reduced to upper or lower triangular form by a QR or LQ factorization and the bdiagonal reduction algorithm is applied to the resulting triangular factor. The singular values of the rectangular rmatrix are then obtained from those of the triangular factor.

If the SVD algorithm did not converge and full accuracy was not attained in the bidiagonal SVD of an intermediate bidiagonal form BD of MAT or in the dqds algorithm applied to BD, function SINGVALUES returns a min(m,n)-vector filled with NAN() function.

For further details, on the SVD of a rectangular matrix and the algorithms to compute it or simply its singular values, see the references (1), (2), (3) or (4). In SINGVALUES function, the reduction to bidiagonal form by orthogonal transformations is parallelized if OPENMP is used, but not the computation of the singular values.

For more informations, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Fernando, K.V., and Parlett, B.N., 1994:** Accurate singular values and differential qd algorithms. Numer. Math., Volume 67, No 2, pp. 191-229.

(4) **Li, S., Gu, M., and Parlett, B. N., 2014:** An Improved DQDS Algorithm. SIAM Journal on Scientific Computing, Volume 36, No 3, C290-C308.

6.19.27 subroutine select_singval_cmp (mat, nsing, s, failure, sort, mul_size, vector, abstol, ls, theta, d, e, tauq, taup, scaling, init, dqds)

Purpose

SELECT_SINGVAL_CMP computes all or some of the greatest singular values of a real m-by-n matrix MAT.

The Singular Value Decomposition (SVD) is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative.

The original matrix MAT is first reduced to upper or lower bidiagonal form BD by an orthogonal transformation:

Q' * MAT * P = BD

where Q and P are orthogonal (see the reference (1) below).

The singular values SIGMA of the bidiagonal matrix BD, which are also the singular values of MAT, are then computed by a bisection algorithm applied to the Tridiagonal Golub-Kahan (TGK) form of the bidiagonal matrix BD (see the reference (2) below, Sec.3.3) or the differential quotient difference with shifts (dqds) algorithm (see the references (3) and (4)).

The routine outputs (parts of) SIGMA and optionally Q and P (in packed form), and BD for a given matrix MAT. SIGMA, Q, P and BD may then be used to obtain selected or all singular vectors of MAT with subroutines BD_INVITER2 or BD_DEFLATE2.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is destroyed and if TAUQ or TAUP are present MAT is overwritten as follows:

- if m >= n, the elements on and below the diagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array TAUP, represent the orthogonal matrix P as a product of elementary reflectors;
- if m < n, the elements below the diagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements on and above the diagonal, with the array TAUP, represent the orthogonal matrix P as a product of elementary reflectors.

See Further Details.

NSING (OUTPUT) integer(i4b) On output, NSING specifies the number of singular values which have been computed. Note that NSING may be greater than the optional argument LS, if multiple singular values at index LS make unique selection impossible and bisection is used.

If none of the optional arguments LS and THETA are used, NSING is set to min(size(MAT,1) , size(MAT,2)) and all the singular values are computed.

S (OUTPUT) real(stnd), dimension(:) On exit, S(1:NSING) contains the first NSING singular values of MAT. The other values in S (S(NSING+1:)) are flagged by a quiet NAN.

The size of S must be min(size(MAT,1) , size(MAT,2)).

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit and the bisection or dqds algorithm converged for all the computed singular values to the desired accuracy ;
- FAILURE = true : indicates that some or all of the singular values failed to converge or were not computed. This is generally caused by unexpectedly inaccurate arithmetic. The sign of the incorrect singular values is set to negative.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MUL_SIZE (INPUT, OPTIONAL) integer(i4b) Internal parameter. MUL_SIZE must verify: 1 <= MUL_SIZE <= n, otherwise a default value is used. For better performance, at the expense of more workspace, a large value can be used.

The default is min(32, n) .

VECTOR (INPUT, OPTIONAL) logical(lgl) On entry, if VECTOR is set to TRUE, a vectorized version of the bisection algorithm is used to compute the singular values SIGMA of the bidiagonal matrix BD.

This argument has no effect if DQDS is equal to true.

The default is VECTOR = false.

ABSTOL (INPUT, OPTIONAL) real(stnd) On entry, the absolute tolerance for the singular values. A singular value (or cluster) is considered to be located in the bisection algorithm if it has has been determined to lie in an interval whose width is ABSTOL or less.

Singular values will be computed most accurately when ABSTOL is set to the square root of the underflow threshold (e.g., sqrt(safmin)=sqrt(LAMCH('S')), not zero.

If ABSTOL is less than or equal to zero, or is not specified, then ULP * | T(GK) | will be used, where | T(GK) | means the 1-norm of the GOLUB-KAHAN tridiagonal form of the bidiagonal matrix BD and ULP is the machine precision (distance from 1 to the next larger floating point number).

This argument has no effect if DQDS is equal to true.

LS (INPUT, OPTIONAL) integer(i4b) On entry, LS specifies the number of singular values which must be computed by the subroutine. On output, NSING may be different than LS if multiple singular values at index LS make unique selection impossible.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

LS must be greater than 0 and less or equal to min(size(MAT,1) , size(MAT,2)).

This argument has no effect if DQDS is equal to true.

The default is LS = min(size(MAT,1) , size(MAT,2)).

THETA (INPUT, OPTIONAL) real(stnd) On entry, THETA specifies that the singular values which are greater or equal to THETA must be computed. If none of the singular values are greater or equal to THETA, NSING is set to zero and S(:) to a quiet NAN.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

This argument has no effect if DQDS is equal to true.

The default is THETA = 0.

D (OUTPUT, OPTIONAL) real(stnd), dimension(:) The diagonal elements of the intermediate bidiagonal matrix BD

The size of D must be min(size(MAT,1) , size(MAT,2)).

E (OUTPUT, OPTIONAL) real(stnd), dimension(:) The off-diagonal elements of the intermediate bidiagonal matrix BD:

- if m >= n, E(i) = BD(i-1,i) for i = 2,3,. . . ,n;
- if m < n, E(i) = BD(i,i-1) for i = 2,3,. . . ,m.

The size of E must be min(size(MAT,1) , size(MAT,2)).

TAUQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix Q. See Further Details.

The size of TAUQ must be min(size(MAT,1) , size(MAT,2)).

TAUP (OUTPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix P. See Further Details.

The size of TAUP must be min(size(MAT,1) , size(MAT,2)).

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the bidiagonal matrix BD is scaled before computing the singular values.

The default is to scale the bidiagonal matrix.

INIT (INPUT, OPTIONAL) logical(lgl) On entry, if INIT=true the initial intervals for the bisection steps are computed from estimates of the eigenvalues of the associated BD' * BD tridiagonal matrix obtained from the Pal-Walker-Kahan algorithm.

This argument has no effect if DQDS is equal to true.

The default is not to use the Pal-Walker-Kahan algorithm.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed.

If DQDS is set to true, all singular values are computed with the dqds algorithm The dqds algorithm is much faster than bisection when all singular values are requested in many cases and delivers about the same accuracy than bisection.

If DQDS is set to false, the requested singular values are computed by bisection.

The default is to use the dqds algorithm if all the singular values are requested and to use bisection if selected singular values are requested (e.g., if the optional arguments LS or THETA are present).

Further Details

The matrices Q and P are represented as products of elementary reflectors:

- If m >= n,

  Q = H(1) * H(2) * . . . * H(n) and P = G(1) * G(2) * . . . * G(n-1)

  Each H(i) and G(i) has the form:

  H(i) = I + tauq * u * u' and G(i) = I + taup * v * v'

  where tauq and taup are real scalars, and u and v are real vectors. Moreover, u(1:i-1) = 0 and v(1:i) = 0.

  If TAUQ or TAUP are present:

  – u(i:m) is stored on exit in MAT(i:m,i);

  – v(i+1:n) is stored on exit in MAT(i,i+1:n).

  If TAUQ is present : tauq is stored in TAUQ(i).

  If TAUP is present : taup is stored in TAUP(i).

- If m < n,

  Q = H(1) * H(2) * . . . * H(m-1) and P = G(1) * G(2) * . . . * G(m)

  Each H(i) and G(i) has the form:

  H(i) = I + tauq * u * u' and G(i) = I + taup * v * v'

where tauq and taup are real scalars, and u and v are real vectors. Moreover, u(1:i) = 0 and v(1:i-1) = 0.

If TAUQ or TAUP are present:

- u(i+1:m) is stored on exit in MAT(i+1:m,i);
- v(i:n) is stored on exit in MAT(i,i:n).

If TAUQ is present : tauq is stored in TAUQ(i).

If TAUP is present : taup is stored in TAUP(i).

The contents of MAT on exit, if TAUQ or TAUP are present, are illustrated by the following examples:

- m = 6 and n = 5 (m >= n):

  (u1 v1 v1 v1 v1)

  (u1 u2 v2 v2 v2)

  (u1 u2 u3 v3 v3)

  (u1 u2 u3 u4 v4)

  (u1 u2 u3 u4 u5)

  (u1 u2 u3 u4 u5)

- m = 5 and n = 6 (m < n):

  (v1 v1 v1 v1 v1 v1)

  (u1 v2 v2 v2 v2 v2)

  (u1 u2 v3 v3 v3 v3)

  (u1 u2 u3 v4 v4 v4)

  (u1 u2 u3 u4 v5 v5)

where ui denotes an element of the vector defining H(i), and vi an element of the vector defining G(i).

Now, let SIGMA(i), i=1,. . . ,N=min(m,n), be the singular values of the intermediate bidiagonal matrix BD in decreasing order of magnitude.

If DQDS is equal to false, the subroutine computes the LS largest singular values (or the singular values which are greater or equal to THETA) of BD by a bisection method (see the reference (1) below, Sec.8.5). The bisection method is applied to an associated 2.N-by-2.N symmetric tridiagonal matrix T (the so-called GOLUB-KAHAN form of BD) whose eigenvalues are the singular values of BD and their negatives (see the reference (2) below).

On the other hand, if DQDS is equal to true, all the singular values of BD are computed by the dqds algorithm (see the references (3) and (4).

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

(3) **Fernando, K.V., and Parlett, B.N., 1994:** Accurate singular values and differential qd algorithms. Numer. Math., Volume 67, No 2, pp. 191-229.

(4) **Li, S., Gu, M., and Parlett, B. N., 2014:** An Improved DQDS Algorithm. SIAM Journal on Scientific Computing, Volume 36, No 3, C290-C308.

6.19.28 subroutine select_singval_cmp (mat, rlmat, nsing, s, failure, sort, mul_size, vector, abstol, ls, theta, d, e, tauo, tauq, taup, scaling, init, dqds)

Purpose

SELECT_SINGVAL_CMP computes all or some of the greatest singular values of a real m-by-n matrix MAT.

The Singular Value Decomposition (SVD) is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative.

The original matrix MAT is first reduced to upper bidiagonal form by a two-step algorithm :

- If m >= n, a QR factorization of the real m-by-n matrix MAT is first computed

  MAT = O * R

where O is orthogonal and R is upper triangular. In a second step, the n-by-n upper triangular matrix R is reduced to upper bidiagonal form BD by an orthogonal transformation :

Q' * R * P = BD

where Q and P are orthogonal and BD is an upper bidiagonal matrix.

- If m < n, an LQ factorization of the real m-by-n matrix MAT is first computed

  MAT = L * O

where O is orthogonal and L is lower triangular. In a second step, the m-by-m lower triangular matrix L is reduced to upper bidiagonal form BD by an orthogonal transformation :

Q' * L * P = BD

where Q and P are orthogonal and BD is an upper bidiagonal matrix.

SELECT_SINGVAL_CMP computes O, BD, Q and P.

The singular values SIGMA of the bidiagonal matrix BD, which are also the singular values of MAT, are then computed by a bisection algorithm applied to the Tridiagonal Golub-Kahan (TGK) form of the bidiagonal matrix BD (see the reference (2) below, Sec.3.3) or the differential quotient difference with shifts (dqds) algorithm (see the references (3) and (4)).

The routine outputs (parts of) SIGMA, and optionally O, Q and P (in packed form), and BD for a given matrix MAT. The matrix O is stored in factored form in the argument MAT if the optional argument TAUO is present or explicitly computed if this argument is absent. SIGMA, O, Q, P and BD may then be used to obtain selected singular vectors of MAT with subroutines BD_INVITER2 or BD_DEFLATE2.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, if:

- m >= n, the elements on and below the diagonal, with the array TAUO, represent the orthogonal matrix O of the QR factorization of MAT, as a product of elementary reflectors, if the argument TAUO is present. Otherwise, the argument MAT contains the first n columns of O on output.
- m < n, the elements on and above the diagonal, with the array TAUO, represent the orthogonal matrix O of the LQ factorization of MAT, as a product of elementary reflectors, if the argument TAUO is present. Otherwise, the argument MAT contains the first m rows of O on output.

See Further Details.

RLMAT (OUTPUT) real(stnd), dimension(:,:)

On exit, the elements on and below the diagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array TAUP, represent the orthogonal matrix P as a product of elementary reflectors;

See Further Details.

The shape of RLMAT must verify: size(RLMAT, 1) = size(RLMAT, 2) = min(size(MAT,1) , size(MAT,2)).

NSING (OUTPUT) integer(i4b) On output, NSING specifies the number of singular values which have been computed. Note that NSING may be greater than the optional argument LS, if multiple singular values at index LS make unique selection impossible.

If none of the optional arguments LS and THETA are used, NSING is set to min(size(MAT,1) , size(MAT,2)) and all the singular values are computed.

S (OUTPUT) real(stnd), dimension(:) On exit, S(1:NSING) contains the first NSING singular values of MAT. The other values in S (S(NSING+1:)) are flagged by a quiet NAN.

The size of S must be min(size(MAT,1) , size(MAT,2)).

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit and the bisection or dqds algorithm converged for all the computed singular values to the desired accuracy ;
- FAILURE = true : indicates that some or all of the singular values failed to converge or were not computed. This is generally caused by unexpectedly inaccurate arithmetic. The sign of the incorrect singular values is set to negative.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MUL_SIZE (INPUT, OPTIONAL) integer(i4b) Internal parameter. MUL_SIZE must verify: 1 <= MUL_SIZE <= n, otherwise a default value is used. For better performance, at the expense of more workspace, a large value can be used.

The default is min(32, n) .

VECTOR (INPUT, OPTIONAL) logical(lgl) On entry, if VECTOR is set to TRUE, a vectorized version of the bisection algorithm is used to compute the singular values SIGMA of the bidiagonal matrix BD.

This argument has no effect if DQDS is equal to true.

The default is VECTOR = false.

ABSTOL (INPUT, OPTIONAL) real(stnd) On entry, the absolute tolerance for the singular values. A singular value (or cluster) is considered to be located in the bisection algorithm if it has has been determined to lie in an interval whose width is ABSTOL or less.

Singular values will be computed most accurately when ABSTOL is set to the square root of the underflow threshold (e.g., sqrt(safmin)=sqrt(LAMCH('S')), not zero.

If ABSTOL is less than or equal to zero, or is not specified, then ULP * | T(GK) | will be used, where | T(GK) | means the 1-norm of the GOLUB-KAHAN tridiagonal form of the bidiagonal matrix BD and ULP is the machine precision (distance from 1 to the next larger floating point number).

This argument has no effect if DQDS is equal to true.

LS (INPUT, OPTIONAL) integer(i4b) On entry, LS specifies the number of singular values which must be computed by the subroutine. On output, NSING may be different than LS if multiple singular values at index LS make unique selection impossible.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

LS must be greater than 0 and less or equal to min(size(MAT,1) , size(MAT,2)).

This argument has no effect if DQDS is equal to true.

The default is LS = min(size(MAT,1) , size(MAT,2)).

THETA (INPUT, OPTIONAL) real(stnd) On entry, THETA specifies that the singular values which are greater or equal to THETA must be computed. If none of the singular values are greater or equal to THETA, NSING is set to zero and S(:) to a quiet NAN.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

This argument has no effect if DQDS is equal to true.

The default is THETA = 0.

D (OUTPUT, OPTIONAL) real(stnd), dimension(:) The diagonal elements of the intermediate bidiagonal matrix BD

The size of D must be min(size(MAT,1) , size(MAT,2)).

E (OUTPUT, OPTIONAL) real(stnd), dimension(:) The off-diagonal elements of the intermediate upper bidiagonal matrix BD:

E(i) = BD(i-1,i) for i = 2,3,. . . ,k;

The size of E must be min(size(MAT,1) , size(MAT,2)).

TAUO (OUTPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix O of the QR or LQ decomposition of MAT.

If the optional argument TAUO is present, the orthogonal matrix O is stored in factored form, as a product of elementary reflectors, in the argument MAT on exit.

If the optional argument TAUO is absent, the orthogonal matrix O is explicitly generated and stored in the argument MAT on exit.

See description of the argument MAT above and Further Details below.

The size of TAUO must be min(size(MAT,1) , size(MAT,2)).

TAUQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix Q. See Further Details.

The size of TAUQ must be min(size(MAT,1) , size(MAT,2)).

TAUP (OUTPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix P. See Further Details.

The size of TAUP must be min(size(MAT,1) , size(MAT,2)).

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the bidiagonal matrix BD is scaled before computing the singular values.

This argument has no effect if DQDS is equal to true.

The default is to scale the bidiagonal matrix.

INIT (INPUT, OPTIONAL) logical(lgl) On entry, if INIT=true the initial intervals for the bisection steps are computed from estimates of the eigenvalues of the associated BD' * BD tridiagonal matrix obtained from the Pal-Walker-Kahan algorithm.

The default is not to use the Pal-Walker-Kahan algorithm.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed.

If DQDS is set to true, all singular values are computed with the dqds algorithm The dqds algorithm is much faster than bisection when all singular values are requested in many cases and delivers about the same accuracy than bisection.

If DQDS is set to false, the requested singular values are computed by bisection.

The default is to use the dqds algorithm if all the singular values are requested and to use bisection if selected singular values are requested (e.g., if the optional arguments LS or THETA are present).

Further Details

If m >= n, the matrix O of the QR factorization of MAT is represented as a product of elementary reflectors

O = W(1) * W(2) * ... * W(n)

Each W(i) has the form

W(i) = I + tauo * (v * v') ,

where tauo is a real scalar and v is a real m-element vector with v(1:i-1) = 0. v(i:m) is stored on exit in MAT(i:m,i) and tauo in TAUO(i). If the optional argument TAUO is absent, the first n columns of O are generated and stored in the argument MAT.

If m < n, The matrix O of the LQ factorization of MAT is represented as a product of elementary reflectors

O = W(m) * ... * W(2) * W(1)

Each W(i) has the form

W(i) = I + tauo * (v * v') ,

where tauo is a real scalar and v is a real n-element vector with v(1:i-1) = 0. v(i:n) is stored on exit in MAT(i,i:n) and tauo in TAUO(i). If the optional argument TAUO is absent, the first m rows of O are generated and stored in the argument MAT.

The matrix O is stored in factored form if the optional argument TAUO is present or explicitly computed if this argument is absent.

A blocked algorithm is used for computing the QR or LQ factorization of MAT. Furthermore, the computations are parallelized if OPENMP is used.

After, the initial QR or LQ factorization of MAT, the (upper or lower) triangular matrix is reduced to upper bidiagonal form BD.

The matrices Q and P of the bidiagonal factorization of the triangular matrix R or L are represented as products of elementary reflectors:

Q = H(1) * H(2) * ... * H(k) and P = G(1) * G(2) * ... * G(k-1)

, where k = min(size(MAT,1) , size(MAT,2)). Each H(i) and G(i) has the form:

H(i) = I + tauq * u * u' and G(i) = I + taup * v * v'

where tauq and taup are real scalars, and u and v are real vectors; u(1:i-1) = 0 and u(i:min(m,n)) is stored on exit in RLMAT(i:min(m,n),i); v(1:i) = 0 and v(i+1:min(m,n)) is stored on exit in RLMAT(i,i+1:min(m,n)); tauq is stored in TAUQ(i) and taup in TAUP(i).

The contents of RLMAT on exit are illustrated by the following example:

m = 6 and n = 5 (m >= n):

(u1 v1 v1 v1 v1)

(u1 u2 v2 v2 v2)

(u1 u2 u3 v3 v3)

(u1 u2 u3 u4 v4)

(u1 u2 u3 u4 u5)

where ui denotes an element of the vector defining H(i), and vi an element of the vector defining G(i).

Now, let SIGMA(i), i=1,...,N=min(m,n), be the singular values of the intermediate bidiagonal matrix BD in decreasing order of magnitude.

If DQDS is equal to false, the subroutine computes the LS largest singular values (or the singular values which are greater or equal to THETA) of BD by a bisection method (see the reference (1) below, Sec.8.5). The bisection method is applied to an associated 2.N-by-2.N symmetric tridiagonal matrix T (the so-called GOLUB-KAHAN form of BD) whose eigenvalues are the singular values of BD and their negatives (see the reference (2) below).

On the other hand, if DQDS is equal to true, all the singular values of BD are computed by the dqds algorithm (see the references (3) and (4).

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

(3) **Fernando, K.V., and Parlett, B.N., 1994:** Accurate singular values and differential qd algorithms. Numer. Math., Volume 67, No 2, pp. 191-229.

(4) **Li, S., Gu, M., and Parlett, B. N., 2014:** An Improved DQDS Algorithm. SIAM Journal on Scientific Computing, Volume 36, No 3, C290-C308.

6.19.29 subroutine select_singval_cmp2 (mat, nsing, s, failure, sort, mul_size, vector, abstol, ls, theta, d, e, tauq, taup, scaling, init, dqds)

Purpose

SELECT_SINGVAL_CMP2 computes all or some of the greatest singular values of a real m-by-n matrix MAT.

The Singular Value Decomposition (SVD) is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative.

The original matrix MAT is first reduced to upper or lower bidiagonal form BD by an orthogonal transformation:

Q' * MAT * P = BD

where Q and P are orthogonal (see the reference (1) below).

The singular values SIGMA of the bidiagonal matrix BD, which are also the singular values of MAT, are then computed by a bisection algorithm (see the reference (1) below, Sec.8.5). The bisection method is applied (implicitly) to the associated min(m,n)-by-min(m,n) symmetric tridiagonal matrix BD' * BD whose eigenvalues are the squares of the singular values of BD by using the differential stationary form of the qd algorithm of Rutishauser (see the reference (2) below, Sec.3.1). Alternatively, at the user option, all singular values can be computed by the differential quotient difference with shifts (dqds) algorithm (see the references (3) and (4)).

The routine outputs (parts of) SIGMA and optionally Q and P (in packed form), and BD for a given matrix MAT. SIGMA, Q, P and BD may then be used to obtain selected or all singular vectors of MAT with subroutines BD_INVITER2 or BD_DEFLATE2.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is destroyed and if TAUQ or TAUP are present MAT is overwritten as follows:

- if m >= n, the elements on and below the diagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array TAUP, represent the orthogonal matrix P as a product of elementary reflectors;
- if m < n, the elements below the diagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements on and above the diagonal, with the array TAUP, represent the orthogonal matrix P as a product of elementary reflectors.

See Further Details.

NSING (OUTPUT) integer(i4b) On output, NSING specifies the number of singular values which have been computed. Note that NSING may be greater than the optional argument LS, if multiple singular values at index LS make unique selection impossible.

If none of the optional arguments LS and THETA are used, NSING is set to min(size(MAT,1) , size(MAT,2)) and all the singular values are computed.

S (OUTPUT) real(stnd), dimension(:) On exit, S(1:NSING) contains the first NSING singular values of MAT. The other values in S (S(NSING+1:)) are flagged by a quiet NAN.

The size of S must be min(size(MAT,1) , size(MAT,2)).

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit and the bisection or dqds algorithm converged for all the computed singular values to the desired accuracy ;
- FAILURE = true : indicates that some or all of the singular values failed to converge or were not computed. This is generally caused by unexpectedly inaccurate arithmetic. The sign of the incorrect singular values is set to negative.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MUL_SIZE (INPUT, OPTIONAL) integer(i4b) Internal parameter. MUL_SIZE must verify: 1 <= MUL_SIZE <= n, otherwise a default value is used. For better performance, at the expense of more workspace, a large value can be used.

The default is min(32, n) .

VECTOR (INPUT, OPTIONAL) logical(lgl) On entry, if VECTOR is set to TRUE, a vectorized version of the bisection algorithm is used to compute the singular values SIGMA of the bidiagonal matrix BD.

This argument has no effect if DQDS is equal to true.

The default is VECTOR = false.

ABSTOL (INPUT, OPTIONAL) real(stnd) On entry, the absolute tolerance for the singular values. A singular value (or cluster) is considered to be located in the bisection algorithm if it has has been determined to lie in an interval whose width is ABSTOL or less.

Singular values will be computed most accurately when ABSTOL is set to the square root of the underflow threshold (e.g., sqrt(safmin)=sqrt(LAMCH('S')), not zero.

If ABSTOL is less than or equal to zero, or is not specified, then ULP * | BD' * BD | will be used, where | BD' * BD | means the 1-norm of the tridiagonal matrix BD' * BD (BD' means the transpose of BD) and ULP is the machine precision (distance from 1 to the next larger floating point number).

This argument has no effect if DQDS is equal to true.

LS (INPUT, OPTIONAL) integer(i4b) On entry, LS specifies the number of singular values which must be computed by the subroutine. On output, NSING may be different than LS if multiple singular values at index LS make unique selection impossible.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

LS must be greater than 0 and less or equal to min(size(MAT,1) , size(MAT,2)).

This argument has no effect if DQDS is equal to true.

The default is LS = min(size(MAT,1) , size(MAT,2)).

THETA (INPUT, OPTIONAL) real(stnd) On entry, THETA specifies that the singular values which are greater or equal to THETA must be computed. If none of the singular values are greater or equal to THETA, NSING is set to zero and S(:) to a quiet NAN.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

This argument has no effect if DQDS is equal to true.

The default is THETA = 0.

D (OUTPUT, OPTIONAL) real(stnd), dimension(:) The diagonal elements of the intermediate bidiagonal matrix BD

The size of D must be min(size(MAT,1) , size(MAT,2)).

E (OUTPUT, OPTIONAL) real(stnd), dimension(:) The off-diagonal elements of the intermediate bidiagonal matrix BD:

- if m >= n, E(i) = BD(i-1,i) for i = 2,3,. . . ,n;
- if m < n, E(i) = BD(i,i-1) for i = 2,3,. . . ,m.

The size of E must be min(size(MAT,1) , size(MAT,2)).

TAUQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix Q. See Further Details.

The size of TAUQ must be min(size(MAT,1) , size(MAT,2)).

TAUP (OUTPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix P. See Further Details.

The size of TAUP must be min(size(MAT,1) , size(MAT,2)).

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the bidiagonal matrix BD is scaled before computing the singular values.

The default is to scale the bidiagonal matrix.

INIT (INPUT, OPTIONAL) logical(lgl) On entry, if INIT=true the initial intervals for the bisection steps are computed from estimates of the eigenvalues of the associated BD' * BD tridiagonal matrix obtained from the Pal-Walker-Kahan algorithm.

This argument has no effect if DQDS is equal to true.

The default is not to use the Pal-Walker-Kahan algorithm.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed.

If DQDS is set to true, all singular values are computed with the dqds algorithm The dqds algorithm is much faster than bisection when all singular values are requested in many cases and delivers about the same accuracy than bisection.

If DQDS is set to false, the requested singular values are computed by bisection.

The default is to use the dqds algorithm if all the singular values are requested and to use bisection if selected singular values are requested (e.g., if the optional arguments LS or THETA are present).

Further Details

The matrices Q and P are represented as products of elementary reflectors:

- If m >= n,

  Q = H(1) * H(2) * . . . * H(n) and P = G(1) * G(2) * . . . * G(n-1)

  Each H(i) and G(i) has the form:

  H(i) = I + tauq * u * u' and G(i) = I + taup * v * v'

  where tauq and taup are real scalars, and u and v are real vectors. Moreover, u(1:i-1) = 0 and v(1:i) = 0.

  If TAUQ or TAUP are present:

  - u(i:m) is stored on exit in MAT(i:m,i);
  - v(i+1:n) is stored on exit in MAT(i,i+1:n).

  If TAUQ is present : tauq is stored in TAUQ(i). If TAUP is present : taup is stored in TAUP(i).

- If m < n,

  Q = H(1) * H(2) * . . . * H(m-1) and P = G(1) * G(2) * . . . * G(m)

  Each H(i) and G(i) has the form:

  H(i) = I + tauq * u * u' and G(i) = I + taup * v * v'

where tauq and taup are real scalars, and u and v are real vectors. Moreover, u(1:i) = 0 and v(1:i-1) = 0.

If TAUQ or TAUP are present:

- u(i+1:m) is stored on exit in MAT(i+1:m,i);
- v(i:n) is stored on exit in MAT(i,i:n).

If TAUQ is present : tauq is stored in TAUQ(i).

If TAUP is present : taup is stored in TAUP(i).

The contents of MAT on exit, if TAUQ or TAUP are present, are illustrated by the following examples:

- m = 6 and n = 5 (m >= n):

  (u1 v1 v1 v1 v1)

  (u1 u2 v2 v2 v2)

  (u1 u2 u3 v3 v3)

  (u1 u2 u3 u4 v4)

  (u1 u2 u3 u4 u5)

  (u1 u2 u3 u4 u5)

- m = 5 and n = 6 (m < n):

  (v1 v1 v1 v1 v1 v1)

  (u1 v2 v2 v2 v2 v2)

  (u1 u2 v3 v3 v3 v3)

  (u1 u2 u3 v4 v4 v4)

  (u1 u2 u3 u4 v5 v5)

where ui denotes an element of the vector defining H(i), and vi an element of the vector defining G(i).

Now, let SIGMA(i), i=1,. . . ,N=min(m,n), be the singular values of the intermediate bidiagonal matrix BD in decreasing order of magnitude.

If DQDS is equal to false, the subroutine computes the LS largest singular values (or the singular values which are greater or equal to THETA) of BD by a bisection method (see the reference (1) below, Sec.8.5). The bisection method is applied (implicitly) to the associated N-by-N symmetric tridiagonal matrix BD' * BD whose eigenvalues are the squares of the singular values of BD by using the differential stationary form of the qd algorithm of Rutishauser (see the reference (2) below, Sec.3.1).

On the other hand, if DQDS is equal to true, all the singular values of BD are computed by the dqds algorithm (see the references (3) and (4).

When bisection is used, SELECT_SINGVAL_CMP2 subroutine is less accurate, but faster, than SELECT_SINGVAL_CMP subroutine since SELECT_SINGVAL_CMP works on the 2.N-by-2.N symmetric tridiagonal GOLUB-KAHAN form of BD, while SELECT_SINGVAL_CMP2 works implicitly on the associated N-by-N symmetric tridiagonal matrix BD' * BD whose eigenvalues are the squares of the singular values of BD.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

(3) **Fernando, K.V., and Parlett, B.N., 1994:** Accurate singular values and differential qd algorithms. Numer. Math., Volume 67, No 2, pp. 191-229.

(4) **Li, S., Gu, M., and Parlett, B. N., 2014:** An Improved DQDS Algorithm. SIAM Journal on Scientific Computing, Volume 36, No 3, C290-C308.

6.19.30 subroutine select_singval_cmp2 (mat, rlmat, nsing, s, failure, sort, mul_size, vector, abstol, ls, theta, d, e, tauo, tauq, taup, scaling, init, dqds)

Purpose

SELECT_SINGVAL_CMP2 computes all or some of the greatest singular values of a real m-by-n matrix MAT.

The Singular Value Decomposition (SVD) is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative.

The original matrix MAT is first reduced to upper bidiagonal form by a two-step algorithm :

- If m >= n, a QR factorization of the real m-by-n matrix MAT is first computed

  MAT = O * R

where O is orthogonal and R is upper triangular. In a second step, the n-by-n upper triangular matrix R is reduced to upper bidiagonal form BD by an orthogonal transformation :

Q' * R * P = BD

where Q and P are orthogonal and BD is an upper bidiagonal matrix.

- If m < n, an LQ factorization of the real m-by-n matrix MAT is first computed

  MAT = L * O

where O is orthogonal and L is lower triangular. In a second step, the m-by-m lower triangular matrix L is reduced to upper bidiagonal form BD by an orthogonal transformation :

Q' * L * P = BD

where Q and P are orthogonal and BD is an upper bidiagonal matrix.

SELECT_SINGVAL_CMP2 computes O, BD, Q and P.

The singular values SIGMA of the bidiagonal matrix BD, which are also the singular values of MAT, are then computed by a bisection algorithm (see the reference (1) below, Sec.8.5). The bisection method is applied (implicitly) to the associated min(m,n)-by-min(m,n) symmetric tridiagonal matrix BD' * BD whose eigenvalues are the squares of the singular values of BD by using the differential stationary form of the qd algorithm of Rutishauser (see the reference (2) below, Sec.3.1). Alternatively, at the user option, all singular values can be computed by the differential quotient difference with shifts (dqds) algorithm (see the references (3) and (4)).

The routine outputs (parts of) SIGMA, and optionally O, Q and P (in packed form), and BD for a given matrix MAT. The matrix O is stored in factored form in the argument MAT if the optional argument TAUO is present or explicitly computed if this argument is absent. SIGMA, O, Q, P and BD may then be used to obtain selected singular vectors of MAT with subroutines BD_INVITER2 or BD_DEFLATE2.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, if:

- m >= n, the elements on and below the diagonal, with the array TAUO, represent the orthogonal matrix O of the QR factorization of MAT, as a product of elementary reflectors, if the argument TAUO is present. Otherwise, the argument MAT contains the first n columns of O on output.
- m < n, the elements on and above the diagonal, with the array TAUO, represent the orthogonal matrix O of the LQ factorization of MAT, as a product of elementary reflectors, if the argument TAUO is present. Otherwise, the argument MAT contains the first m rows of O on output.

See Further Details.

RLMAT (OUTPUT) real(stnd), dimension(:,:)

On exit, the elements on and below the diagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array TAUP, represent the orthogonal matrix P as a product of elementary reflectors;

See Further Details.

The shape of RLMAT must verify: size(RLMAT, 1) = size(RLMAT, 2) = min(size(MAT,1) , size(MAT,2)).

NSING (OUTPUT) integer(i4b) On output, NSING specifies the number of singular values which have been computed. Note that NSING may be greater than the optional argument LS, if multiple singular values at index LS make unique selection impossible.

If none of the optional arguments LS and THETA are used, NSING is set to min(size(MAT,1) , size(MAT,2)) and all the singular values are computed.

S (OUTPUT) real(stnd), dimension(:) On exit, S(1:NSING) contains the first NSING singular values of MAT. The other values in S (S(NSING+1:)) are flagged by a quiet NAN.

The size of S must be min(size(MAT,1) , size(MAT,2)).

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit and the bisection algorithm converged for all the computed singular values to the desired accuracy ;
- FAILURE = true : indicates that some or all of the singular values failed to converge or were not computed. This is generally caused by unexpectedly inaccurate arithmetic. The sign of the incorrect singular values is set to negative.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MUL_SIZE (INPUT, OPTIONAL) integer(i4b) Internal parameter. MUL_SIZE must verify: 1 <= MUL_SIZE <= n, otherwise a default value is used. For better performance, at the expense of more workspace, a large value can be used.

The default is min(32, n) .

VECTOR (INPUT, OPTIONAL) logical(lgl) On entry, if VECTOR is set to TRUE, a vectorized version of the bisection algorithm is used to compute the singular values SIGMA of the bidiagonal matrix BD.

This argument has no effect if DQDS is equal to true.

The default is VECTOR = false.

ABSTOL (INPUT, OPTIONAL) real(stnd) On entry, the absolute tolerance for the singular values. A singular value (or cluster) is considered to be located in the bisection algorithm if it has has been determined to lie in an interval whose width is ABSTOL or less.

Singular values will be computed most accurately when ABSTOL is set to the square root of the underflow threshold, sqrt(LAMCH('S')), not zero.

If ABSTOL is less than or equal to zero, or is not specified, then ULP * | BD' * BD | will be used, where | BD' * BD | means the 1-norm of the tridiagonal matrix BD' * BD (BD' means the transpose of BD) and ULP is the machine precision (distance from 1 to the next larger floating point number).

LS (INPUT, OPTIONAL) integer(i4b) On entry, LS specifies the number of singular values which must be computed by the subroutine. On output, NSING may be different than LS if multiple singular values at index LS make unique selection impossible.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

LS must be greater than 0 and less or equal to min(size(MAT,1) , size(MAT,2)).

This argument has no effect if DQDS is equal to true.

The default is LS = min(size(MAT,1) , size(MAT,2)).

THETA (INPUT, OPTIONAL) real(stnd) On entry, THETA specifies that the singular values which are greater or equal to THETA must be computed. If none of the singular values are greater or equal to THETA, NSING is set to zero and S(:) to a quiet NAN.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

This argument has no effect if DQDS is equal to true.

The default is THETA = 0.

D (OUTPUT, OPTIONAL) real(stnd), dimension(:) The diagonal elements of the intermediate bidiagonal matrix BD

The size of D must be min(size(MAT,1) , size(MAT,2)).

E (OUTPUT, OPTIONAL) real(stnd), dimension(:) The off-diagonal elements of the intermediate upper bidiagonal matrix BD:

E(i) = BD(i-1,i) for i = 2,3,. . . ,k;

The size of E must be min(size(MAT,1) , size(MAT,2)).

TAUO (OUTPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix O of the QR or LQ decomposition of MAT.

If the optional argument TAUO is present, the orthogonal matrix O is stored in factored form, as a product of elementary reflectors, in the argument MAT on exit.

If the optional argument TAUO is absent, the orthogonal matrix O is explicitly generated and stored in the argument MAT on exit.

See description of the argument MAT above and Further Details below.

The size of TAUO must be min(size(MAT,1) , size(MAT,2)).

TAUQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix Q. See Further Details.

The size of TAUQ must be min(size(MAT,1) , size(MAT,2)).

TAUP (OUTPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix P. See Further Details.

The size of TAUP must be min(size(MAT,1) , size(MAT,2)).

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the bidiagonal matrix BD is scaled before computing the singular values.

The default is to scale the bidiagonal matrix.

INIT (INPUT, OPTIONAL) logical(lgl) On entry, if INIT=true the initial intervals for the bisection steps are computed from estimates of the eigenvalues of the associated BD' * BD tridiagonal matrix obtained from the Pal-Walker-Kahan algorithm.

This argument has no effect if DQDS is equal to true.

The default is not to use the Pal-Walker-Kahan algorithm.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed.

If DQDS is set to true, all singular values are computed with the dqds algorithm The dqds algorithm is much faster than bisection when all singular values are requested in many cases and delivers about the same accuracy than bisection.

If DQDS is set to false, the requested singular values are computed by bisection.

The default is to use the dqds algorithm if all the singular values are requested and to use bisection if selected singular values are requested (e.g., if the optional arguments LS or THETA are present).

Further Details

If m >= n, the matrix O of the QR factorization of MAT is represented as a product of elementary reflectors

O = W(1) * W(2) * ... * W(n)

Each W(i) has the form

W(i) = I + tauo * (v * v') ,

where tauo is a real scalar and v is a real m-element vector with v(1:i-1) = 0. v(i:m) is stored on exit in MAT(i:m,i) and tauo in TAUO(i). If the optional argument TAUO is absent, the first n columns of O are generated and stored in the argument MAT.

If m < n, The matrix O of the LQ factorization of MAT is represented as a product of elementary reflectors

O = W(m) * ... * W(2) * W(1)

Each W(i) has the form

W(i) = I + tauo * (v * v') ,

where tauo is a real scalar and v is a real n-element vector with v(1:i-1) = 0. v(i:n) is stored on exit in MAT(i,i:n) and tauo in TAUO(i). If the optional argument TAUO is absent, the first m rows of O are generated and stored in the argument MAT.

The matrix O is stored in factored form if the optional argument TAUO is present or explicitly computed if this argument is absent.

A blocked algorithm is used for computing the QR or LQ factorization of MAT. Furthermore, the computations are parallelized if OPENMP is used.

After, the initial QR or LQ factorization of MAT, the (upper or lower) triangular matrix is reduced to upper bidiagonal form BD.

The matrices Q and P of the bidiagonal factorization of the triangular matrix R or L are represented as products of elementary reflectors:

Q = H(1) * H(2) * ... * H(k) and P = G(1) * G(2) * ... * G(k-1)

, where k = min(size(MAT,1) , size(MAT,2)). Each H(i) and G(i) has the form:

H(i) = I + tauq * u * u' and G(i) = I + taup * v * v'

where tauq and taup are real scalars, and u and v are real vectors; u(1:i-1) = 0 and u(i:min(m,n)) is stored on exit in RLMAT(i:min(m,n),i); v(1:i) = 0 and v(i+1:min(m,n)) is stored on exit in RLMAT(i,i+1:min(m,n)); tauq is stored in TAUQ(i) and taup in TAUP(i).

The contents of RLMAT on exit are illustrated by the following example:

m = 6 and n = 5 (m >= n):

(u1 v1 v1 v1 v1)

(u1 u2 v2 v2 v2)

(u1 u2 u3 v3 v3)

(u1 u2 u3 u4 v4)

(u1 u2 u3 u4 u5)

where ui denotes an element of the vector defining H(i), and vi an element of the vector defining G(i).

Now, let SIGMA(i), i=1,...,N=min(m,n), be the singular values of the intermediate bidiagonal matrix BD in decreasing order of magnitude.

If DQDS is equal to false, the subroutine computes the LS largest singular values (or the singular values which are greater or equal to THETA) of BD by a bisection method (see the reference (1) below, Sec.8.5). The bisection method is applied (implicitly) to the associated N-by-N symmetric tridiagonal matrix BD' * BD whose eigenvalues are the squares of the singular values of BD by using the differential stationary form of the qd algorithm of Rutishauser (see the reference (2) below, Sec.3.1).

On the other hand, if DQDS is equal to true, all the singular values of BD are computed by the dqds algorithm (see the references (3) and (4).

When bisection is used, SELECT_SINGVAL_CMP2 subroutine is less accurate, but faster, than SELECT_SINGVAL_CMP subroutine since SELECT_SINGVAL_CMP works on the 2.N-by-2.N symmetric tridiagonal GOLUB-KAHAN form of BD, while SELECT_SINGVAL_CMP2 works implicitly on the associated N-by-N symmetric tridiagonal matrix BD' * BD whose eigenvalues are the squares of the singular values of BD.

For further details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

(3) **Fernando, K.V., and Parlett, B.N., 1994:** Accurate singular values and differential qd algorithms. Numer. Math., Volume 67, No 2, pp. 191-229.

(4) **Li, S., Gu, M., and Parlett, B. N., 2014:** An Improved DQDS Algorithm. SIAM Journal on Scientific Computing, Volume 36, No 3, C290-C308.

6.19.31 subroutine select_singval_cmp3 (mat, nsing, s, failure, sort, mul_size, vector, abstol, ls, theta, d, e, p, gen_p, reortho, scaling, init, dqds, failure_bd)

Purpose

SELECT_SINGVAL_CMP3 computes all or some of the greatest singular values of a real m-by-n matrix MAT with m>=n.

The Singular Value Decomposition (SVD) is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative.

The original matrix MAT is first reduced to upper bidiagonal form BD by an orthogonal transformation:

Q' * MAT * P = BD

where Q and P are orthogonal (see the reference (5) below). The Ralha-Barlow one-sided method is used for this purpose (see the references (1) to (3) below).

The singular values SIGMA of the bidiagonal matrix BD, which are also the singular values of MAT, are then computed by a bisection algorithm applied to the Tridiagonal Golub-Kahan (TGK) of the bidiagonal matrix BD (see the reference (6) below, Sec.3.3) or the differential quotient difference with shifts (dqds) algorithm (see the references (7) and (8)).

The routine outputs (parts of) SIGMA, Q and optionally P (in packed form) and BD for a given matrix MAT. SIGMA, Q, P and BD may then be used to obtain selected singular vectors of MAT with subroutines BD_INVITER2 or BD_DEFLATE2.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is overwritten with the first n columns of Q (stored column-wise), the orthogonal matrix used to reduce MAT to bidiagonal form as returned by subroutine BD_CMP2 in its argument MAT.

The shape of MAT must verify: size(MAT, 1) >= size(MAT, 2) = n .

NSING (OUTPUT) integer(i4b) On output, NSING specifies the number of singular values which have been computed. Note that NSING may be greater than the optional argument LS, if multiple singular values at index LS make unique selection impossible.

If none of the optional arguments LS and THETA are used, NSING is set to size(MAT,2) and all the singular values are computed.

S (OUTPUT) real(stnd), dimension(:) On exit, S(1:NSING) contains the first NSING singular values of MAT. The other values in S (S(NSING+1:)) are flagged by a quiet NAN.

The size of S must be equal to size(MAT, 2) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit and the bisection or dqds algorithm converged for all the computed singular values to the desired accuracy ;
- FAILURE = true : indicates that some or all of the singular values failed to converge or were not computed. This is generally caused by unexpectedly inaccurate arithmetic. The sign of the incorrect singular values is set to negative.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MUL_SIZE (INPUT, OPTIONAL) integer(i4b) Internal parameter. MUL_SIZE must verify: 1 <= MUL_SIZE <= n, otherwise a default value is used. For better performance, at the expense of more workspace, a large value can be used.

The default is min(32, n).

VECTOR (INPUT, OPTIONAL) logical(lgl) On entry, the absolute tolerance for the singular values. A singular value (or cluster) is considered to be located in the bisection algorithm if it has has been determined to lie in an interval whose width is ABSTOL or less.

This argument has no effect if DQDS is equal to true.

The default is VECTOR = false.

ABSTOL (INPUT, OPTIONAL) real(stnd) On entry, the absolute tolerance for the singular values. A singular value (or cluster) is considered to be located in the bisection algorithm if it been determined to lie in an interval whose width is ABSTOL or less.

Singular values will be computed most accurately when ABSTOL is set to the square root of the underflow threshold (e.g., sqrt(safmin)=sqrt(LAMCH('S')), not zero.

If ABSTOL is less than or equal to zero, or is not specified, then ULP * | T(GK) | will be used, where | T(GK) | means the 1-norm of the GOLUB-KAHAN tridiagonal form of the bidiagonal matrix BD and ULP is the machine precision (distance from 1 to the next larger floating point number).

This argument has no effect if DQDS is equal to true.

LS (INPUT, OPTIONAL) integer(i4b) On entry, LS specifies the number of singular values which must be computed by the subroutine. On output, NSING may be different than LS if multiple singular values at index LS make unique selection impossible.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

LS must be greater than 0 and less or equal to size(MAT, 2) = n .

This argument has no effect if DQDS is equal to true.

The default is LS = size(MAT, 2) = n .

THETA (INPUT, OPTIONAL) real(stnd) On entry, THETA specifies that the singular values which are greater or equal to THETA must be computed. If none of the singular values are greater or equal to THETA, NSING is set to zero and S(:) to a quiet NAN.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

This argument has no effect if DQDS is equal to true.

The default is THETA = 0 .

D (OUTPUT, OPTIONAL) real(stnd), dimension(:) The diagonal elements of the intermediate bidiagonal matrix BD

The size of D must be equal to size(MAT, 2) = n .

E (OUTPUT, OPTIONAL) real(stnd), dimension(:) The off-diagonal elements of the intermediate upper bidiagonal matrix BD:

E(i) = B(i-1,i) for i = 2,3,. . . ,n;

E(1) is arbitrary.

The size of E must be equal to size(MAT, 2) = n .

P (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, P is overwritten with the n-by-n orthogonal matrix P (stored column-wise or in packed form), the orthogonal matrix used to reduce MAT to bidiagonal form as returned by subroutine BD_CMP2 in its argument P.

The shape of P must verify: size(P, 1) = size(P, 2) = n .

GEN_P (INPUT, OPTIONAL) logical(lgl) On entry, this optional argument has an effect only if the optional argument P is also used.

In this case, if the optional argument GEN_P is used and is set to true, the orthogonal matrix P used to reduce MAT to bidiagonal form is generated on output of the subroutine in its argument P.

If GEN_P is set to false, the orthogonal matrix P is stored in factored form as products of elementary reflectors in the lower triangle of the array P.

See the description of BD_CMP2 subroutine for more details.

The default is true.

REORTHO (INPUT, OPTIONAL) logical(lgl) If the optional argument REORTHO is set to true, reorthogonalization of the matrix Q, used to reduce MAT to bidiagonal form, is performed (when needed). If this optional logical argument is set to false, reorthogonalization of Q is never performed and the columns of the matrix Q can be far from orthogonal to each other for input matrices MAT with a large condition number.

See the description of BD_CMP2 subroutine for more details.

The default is REORTHO = false.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the bidiagonal matrix BD is scaled before computing the singular values.

The default is to scale the bidiagonal matrix.

INIT (INPUT, OPTIONAL) logical(lgl) On entry, if INIT=true the initial intervals for the bisection steps are computed from estimates of the eigenvalues of the associated BD' * BD tridiagonal matrix obtained from the Pal-Walker-Kahan algorithm.

This argument has no effect if DQDS is equal to true.

The default is not to use the Pal-Walker-Kahan algorithm.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed.

If DQDS is set to true, all singular values are computed with the dqds algorithm The dqds algorithm is much faster than bisection when all singular values are requested in many cases and delivers about the same accuracy than bisection.

If DQDS is set to false, the requested singular values are computed by bisection.

The default is to use the dqds algorithm if all the singular values are requested and to use bisection if selected singular values are requested (e.g., if the optional arguments LS or THETA are present).

FAILURE_BD (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE_BD = false : indicates that maximum accuracy was obtained in the Ralha-Barlow one-sided bidiagonalization of MAT.

- FAILURE_BD = true : indicates that MAT is nearly singular and some loss of orthogonality can be expected in the Ralha-Barlow bidiagonalization algorithm.

See description of the BD_CMP2 subroutine for further details.

Further Details

The matrices Q, P and BD are computed with the help of the Ralha-Barlow one-sided method. Q is computed by a recurrence relationship and the first n columns of Q are stored in the argument MAT on exit. P is computed as a product of n-1 elementary reflectors (e.g. Householder transformations):

P = G(1) * G(2) * ... * G(n-1)

Each G(i) has the form:

G(i) = I + taup * v * v'

where taup is a real scalar, and v is a real vector. IF GEN_P is used and set to false, the n-1 G(i) elementary reflectors are stored in the lower triangle of the array P.

For the G(i) reflector, taup is stored in P(i+1,1) and v is stored in P(i+1:n,i+1). In addition, P(1,1) is set to -1 if GEN_P=false and is equal to 1 if GEN_P=true.

In other words, the value of P(1,1) indicates if the orthogonal matrix P is stored in factored form or not. Note that if n is equal to 1, elementary reflectors are not needed and consequently P(1,1) is set to 1, independently of the value of GEN_P.

This is the blocked version of the Ralha-Barlow one-sided algorithm. See the references (1), (2) and (3) for further details. Furthermore, the algorithm is parallelized if OPENMP is used.

Since Q is computed by a recurrence relationship, a loss of orthogonality of Q can be observed when the rectangular matrix MAT is singular or nearly singular or has a large condition number, see the reference (2) for details.

To correct partly this deficiency, partial reorthogonalization can be performed to ensure orthogonality at the expense of speed of computation. The reorthogonalization uses the Gram-Schmidt method described in the reference (4) and is activated if the optional logical argument REORTHO is set to true.

The reference (2) also explains how to handle the case of an exactly singular matrix MAT (a very rare event). However, in this subroutine, the partial reorthogonalization described in the reference (4), which is used when REORTHO is set true, corrects automatically this problem.

Now, let SIGMA(i), i=1,...,n, be the singular values of the intermediate bidiagonal matrix BD in decreasing order of magnitude.

If DQDS is equal to false, the subroutine computes the LS largest singular values (or the singular values which are greater or equal to THETA) of BD by a bisection method (see the reference (5) below, Sec.8.5). The bisection method is applied to an associated 2.n-by-2.n symmetric tridiagonal matrix T (the so-called GOLUB-KAHAN form of BD) whose eigenvalues are the singular values of BD and their negatives (see the reference (6) below).

On the other hand, if DQDS is equal to true, all the singular values of BD are computed by the dqds algorithm (see the references (7) and (8).

For further details, see:

(1) **Ralha, R.M.S., 2003:** One-sided reduction to bidiagonal form. Linear Algebra Appl., No 358, pp. 219-238.

(2) **Barlow, J.L., Bosner, N., and Drmac, Z., 2005:** A new stable bidiagonal reduction algorithm. Linear Algebra Appl., No 397, pp. 35-84.

(3) **Bosner, N., and Barlow, J.L., 2007:** Block and Parallel versions of one-sided bidiagonalization. SIAM J. Matrix Anal. Appl., Volume 29, No 3, pp. 927-953.

(4) **Stewart, G.W., 2007:** Block Gram-Schmidt Orthogonalization. Report TR-4823, Department of Computer Science, College Park, University of Maryland.

(5) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(6) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

(7) **Fernando, K.V., and Parlett, B.N., 1994:** Accurate singular values and differential qd algorithms. Numer. Math., Volume 67, No 2, pp. 191-229.

(8) **Li, S., Gu, M., and Parlett, B. N., 2014:** An Improved DQDS Algorithm. SIAM Journal on Scientific Computing, Volume 36, No 3, C290-C308.

6.19.32 subroutine select_singval_cmp3 (mat, rmat, nsing, s, failure, sort, mul_size, vector, abstol, ls, theta, d, e, tauo, p, gen_p, reortho, scaling, init, dqds, failure_bd)

Purpose

SELECT_SINGVAL_CMP3 computes all or some of the greatest singular values of a real m-by-n matrix MAT with m>=n.

The Singular Value Decomposition (SVD) is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative.

The original matrix MAT is first reduced to upper bidiagonal form by a two-step algorithm :

A QR factorization of the real m-by-n matrix MAT is first computed

MAT = O * R

where O is orthogonal and R is upper triangular.

In a second step, the n-by-n upper triangular matrix R is reduced to upper bidiagonal form BD by an orthogonal transformation :

Q' * R * P = BD

where Q and P are orthogonal and BD is an upper bidiagonal matrix (see the reference (5) below). The Ralha-Barlow one-sided method is used in this second step (see the references (1) to (3) below).

SELECT_SINGVAL_CMP3 computes O, BD, Q and P.

The singular values SIGMA of the bidiagonal matrix BD, which are also the singular values of MAT, are then computed by a bisection algorithm applied to the Tridiagonal Golub-Kahan (TGK) of the bidiagonal matrix BD (see the reference (6) below, Sec.3.3) or the differential quotient difference with shifts (dqds) algorithm (see the references (7) and (8)).

The routine outputs (parts of) SIGMA, and optionally O, Q and P and BD for a given matrix MAT. The matrix O is stored in factored form in the argument MAT if the optional argument TAUO is present or explicitly computed if this argument is absent. The first n columns of Q are stored in the argument

RMAT. Finally, P is stored in the optional argument P. P is stored in factored form or explicitly generated depending on the value of the optional logical argument GEN_P.

SIGMA, O, Q, P and BD may then be used to obtain selected singular vectors of MAT with subroutines BD_INVITER2 or BD_DEFLATE2.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, the elements on and below the diagonal, with the array TAUO, represent the orthogonal matrix O of the QR factorization of MAT, as a product of elementary reflectors, if the argument TAUO is present. Otherwise, the argument MAT contains the first n columns of O (stored column-wise) on output.

See Further Details.

The shape of MAT must verify: size(MAT, 1) >= size(MAT, 2) = n .

RMAT (OUTPUT) real(stnd), dimension(:,:)

On exit, RMAT contains the first n columns of Q (stored column-wise), the orthogonal matrix used to reduce R to bidiagonal form as returned by subroutine BD_CMP2 in its argument MAT.

See Further Details.

The shape of RMAT must verify: size(RMAT, 1) = size(RMAT, 2) = size(MAT, 2) = n .

NSING (OUTPUT) integer(i4b) On output, NSING specifies the number of singular values which have been computed. Note that NSING may be greater than the optional argument LS, if multiple singular values at index LS make unique selection impossible.

If none of the optional arguments LS and THETA are used, NSING is set to size(MAT,2) and all the singular values are computed.

S (OUTPUT) real(stnd), dimension(:) On exit, S(1:NSING) contains the first NSING singular values of MAT. The other values in S (S(NSING+1:)) are flagged by a quiet NAN.

The size of S must be equal to size(MAT, 2) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit and the bisection or dqds algorithm converged for all the computed singular values to the desired accuracy ;
- FAILURE = true : indicates that some or all of the singular values failed to converge or were not computed. This is generally caused by unexpectedly inaccurate arithmetic. The sign of the incorrect singular values is set to negative.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MUL_SIZE (INPUT, OPTIONAL) integer(i4b) Internal parameter. MUL_SIZE must verify: 1 <= MUL_SIZE <= n, otherwise a default value is used. For better performance, at the expense of more workspace, a large value can be used.

The default is min(32, n).

VECTOR (INPUT, OPTIONAL) logical(lgl) On entry, if VECTOR is set to TRUE, a vectorized version of the bisection algorithm is used to compute the singular values SIGMA of the bidiagonal matrix BD.

This argument has no effect if DQDS is equal to true.

The default is VECTOR = false.

ABSTOL (INPUT, OPTIONAL) real(stnd) On entry, the absolute tolerance for the singular values. A singular value (or cluster) is considered to be located in the bisection algorithm if it been determined to lie in an interval whose width is ABSTOL or less.

Singular values will be computed most accurately when ABSTOL is set to the square root of the underflow threshold (e.g., sqrt(safmin)=sqrt(LAMCH('S')), not zero.

If ABSTOL is less than or equal to zero, or is not specified, then ULP * | T(GK) | will be used, where | T(GK) | means the 1-norm of the GOLUB-KAHAN tridiagonal form of the bidiagonal matrix BD and ULP is the machine precision (distance from 1 to the next larger floating point number).

This argument has no effect if DQDS is equal to true.

LS (INPUT, OPTIONAL) integer(i4b) On entry, LS specifies the number of singular values which must be computed by the subroutine. On output, NSING may be different than LS if multiple singular values at index LS make unique selection impossible.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

LS must be greater than 0 and less or equal to size(MAT, 2) = n .

This argument has no effect if DQDS is equal to true.

The default is LS = size(MAT, 2) = n .

THETA (INPUT, OPTIONAL) real(stnd) On entry, THETA specifies that the singular values which are greater or equal to THETA must be computed. If none of the singular values are greater or equal to THETA, NSING is set to zero and S(:) to a quiet NAN.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

This argument has no effect if DQDS is equal to true.

The default is THETA = 0 .

D (OUTPUT, OPTIONAL) real(stnd), dimension(:) The diagonal elements of the intermediate bidiagonal matrix BD

The size of D must be equal to size(MAT, 2) = n .

E (OUTPUT, OPTIONAL) real(stnd), dimension(:) The off-diagonal elements of the intermediate upper bidiagonal matrix BD:

E(i) = B(i-1,i) for i = 2,3,. . . ,n;

E(1) is arbitrary.

The size of E must be equal to size(MAT, 2) = n .

TAUO (OUTPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix O of the QR decomposition of MAT.

If the optional argument TAUO is present, the orthogonal matrix O is stored in factored form, as a product of elementary reflectors, in the argument MAT on exit.

If the optional argument TAUO is absent, the first n columns of the orthogonal matrix O are explicitly generated and stored in the argument MAT on exit.

See description of the argument MAT above and Further Details below.

The size of TAUO must be equal to size(MAT, 2) = n .

P (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, P is overwritten with the n-by-n orthogonal matrix P (stored column-wise or in packed form), the orthogonal matrix used to reduce MAT to bidiagonal form as returned by subroutine BD_CMP2 in its argument P.

The shape of P must verify: size(P, 1) = size(P, 2) = n .

GEN_P (INPUT, OPTIONAL) logical(lgl) On entry, this optional argument has an effect only if the optional argument P is also used.

In this case, if the optional argument GEN_P is used and is set to true, the orthogonal matrix P used to reduce MAT to bidiagonal form is generated on output of the subroutine in its argument P.

If GEN_P is set to false, the orthogonal matrix P is stored in factored form as products of elementary reflectors in the lower triangle of the array P.

See the description of BD_CMP2 subroutine for more details.

The default is true.

REORTHO (INPUT, OPTIONAL) logical(lgl) If the optional argument REORTHO is set to true, reorthogonalization of the matrix Q, used to reduce MAT to bidiagonal form, is performed (when needed). If this optional logical argument is set to false, reorthogonalization of Q is never performed and the columns of the matrix Q can be far from orthogonal to each other for input matrices MAT with a large condition number.

See the description of BD_CMP2 subroutine for more details.

The default is REORTHO = false.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the bidiagonal matrix BD is scaled before computing the singular values.

The default is to scale the bidiagonal matrix.

INIT (INPUT, OPTIONAL) logical(lgl) On entry, if INIT=true the initial intervals for the bisection steps are computed from estimates of the eigenvalues of the associated BD' * BD tridiagonal matrix obtained from the Pal-Walker-Kahan algorithm.

This argument has no effect if DQDS is equal to true.

The default is not to use the Pal-Walker-Kahan algorithm.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed.

If DQDS is set to true, all singular values are computed with the dqds algorithm The dqds algorithm is much faster than bisection when all singular values are requested in many cases and delivers about the same accuracy than bisection.

If DQDS is set to false, the requested singular values are computed by bisection.

The default is to use the dqds algorithm if all the singular values are requested and to use bisection if selected singular values are requested (e.g., if the optional arguments LS or THETA are present).

FAILURE_BD (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE_BD = false : indicates that maximum accuracy was obtained in the Ralha-Barlow one-sided bidiagonalization of MAT.
- FAILURE_BD = true : indicates that MAT is nearly singular and some loss of orthogonality can be expected in the Ralha-Barlow bidiagonalization algorithm.

See description of the BD_CMP2 subroutine for further details.

Further Details

the matrix O of the QR factorization of MAT is represented as a product of elementary reflectors

O = W(1) * W(2) * ... * W(n)

Each W(i) has the form

W(i) = I + tauo * (v * v') ,

where tauo is a real scalar and v is a real m-element vector with v(1:i-1) = 0. v(i:m) is stored on exit in MAT(i:m,i) and tauo in TAUO(i). If the optional argument TAUO is absent, the first n columns of O are generated and stored in the argument MAT.

The matrix O is stored in factored form if the optional argument TAUO is present or explicitly computed if this argument is absent.

A blocked algorithm is used for computing the QR factorization of MAT. Furthermore, the computations are parallelized if OPENMP is used.

After, the initial QR factorization of MAT, the upper triangular matrix R is reduced to upper bidiagonal form BD:

Q' * R * P = BD

The matrices Q, P and BD are computed with the help of the Ralha-Barlow one-sided method. Q is computed by a recurrence relationship and the first n columns of Q are stored in the argument RMAT on exit. P is computed as a product of n-1 elementary reflectors (e.g. Householder transformations):

P = G(1) * G(2) * ... * G(n-1)

Each G(i) has the form:

G(i) = I + taup * v * v'

where taup is a real scalar, and v is a real vector. IF GEN_P is used and set to false, the n-1 G(i) elementary reflectors are stored in the lower triangle of the array P.

For the G(i) reflector, taup is stored in P(i+1,1) and v is stored in P(i+1:n,i+1). In addition, P(1,1) is set to -1 if GEN_P=false and is equal to 1 if GEN_P=true.

In other words, the value of P(1,1) indicates if the orthogonal matrix P is stored in factored form or not. Note that if n is equal to 1, elementary reflectors are not needed and consequently P(1,1) is set to 1, independently of the value of GEN_P.

This is the blocked version of the Ralha-Barlow one-sided algorithm. See the references (1), (2) and (3) for further details. Furthermore, the algorithm is parallelized if OPENMP is used.

Since Q is computed by a recurrence relationship, a loss of orthogonality of Q can be observed when the rectangular matrix MAT is singular or nearly singular or has a large condition number, see the reference (2) for details.

To correct partly this deficiency, partial reorthogonalization can be performed to ensure orthogonality at the expense of speed of computation. The reorthogonalization uses the Gram-Schmidt method described in the reference (4) and is activated if the optional logical argument REORTHO is set to true.

The reference (2) also explains how to handle the case of an exactly singular matrix MAT (a very rare event). However, in this subroutine, the partial reorthogonalization described in the reference (4), which is used when REORTHO is set true, corrects automatically this problem.

Now, let SIGMA(i), i=1,...,n, be the singular values of the intermediate bidiagonal matrix BD in decreasing order of magnitude.

If DQDS is equal to false, the subroutine computes the LS largest singular values (or the singular values which are greater or equal to THETA) of BD by a bisection method (see the reference (5) below, Sec.8.5). The bisection method is applied to an associated 2.n-by-2.n symmetric tridiagonal matrix T (the so-called GOLUB-KAHAN form of BD) whose eigenvalues are the singular values of BD and their negatives (see the reference (6) below).

On the other hand, if DQDS is equal to true, all the singular values of BD are computed by the dqds algorithm (see the references (7) and (8).

For further details, see:

(1) **Ralha, R.M.S., 2003:** One-sided reduction to bidiagonal form. Linear Algebra Appl., No 358, pp. 219-238.

(2) **Barlow, J.L., Bosner, N., and Drmac, Z., 2005:** A new stable bidiagonal reduction algorithm. Linear Algebra Appl., No 397, pp. 35-84.

(3) **Bosner, N., and Barlow, J.L., 2007:** Block and Parallel versions of one-sided bidiagonalization. SIAM J. Matrix Anal. Appl., Volume 29, No 3, pp. 927-953.

(4) **Stewart, G.W., 2007:** Block Gram-Schmidt Orthogonalization. Report TR-4823, Department of Computer Science, College Park, University of Maryland.

(5) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(6) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

(7) **Fernando, K.V., and Parlett, B.N., 1994:** Accurate singular values and differential qd algorithms. Numer. Math., Volume 67, No 2, pp. 191-229.

(8) **Li, S., Gu, M., and Parlett, B. N., 2014:** An Improved DQDS Algorithm. SIAM Journal on Scientific Computing, Volume 36, No 3, C290-C308.

6.19.33 subroutine select_singval_cmp4 (mat, nsing, s, failure, sort, mul_size, vector, abstol, ls, theta, d, e, p, gen_p, reortho, scaling, init, dqds, failure_bd)

Purpose

SELECT_SINGVAL_CMP4 computes all or some of the greatest singular values of a real m-by-n matrix MAT with m>=n.

The Singular Value Decomposition (SVD) is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative.

The original matrix MAT is first reduced to upper bidiagonal form BD by an orthogonal transformation:

Q' * MAT * P = BD

where Q and P are orthogonal (see the reference (5) below). The Ralha-Barlow one-sided method is used for this purpose (see the references (1) to (3) below).

The singular values SIGMA of the bidiagonal matrix BD, which are also the singular values of MAT, are then computed by a bisection algorithm (see the reference (5) below, Sec.8.5). The bisection method is applied (implicitly) to the associated n-by-n symmetric tridiagonal matrix BD' * BD whose eigenvalues are the squares of the singular values of BD by using the differential stationary form of the qd algorithm of Rutishauser (see the reference (6) below, Sec.3.1). Alternatively, at the user option, all singular values can be computed by the differential quotient difference with shifts (dqds) algorithm (see the references (7) and (8)).

The routine outputs (parts of) SIGMA, Q and optionally P (in packed form) and BD for a given matrix MAT. SIGMA, Q, P and BD may then be used to obtain selected singular vectors of MAT with subroutines BD_INVITER2 or BD_DEFLATE2.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is overwritten with the first n columns of Q (stored column-wise), the orthogonal matrix used to reduce MAT to bidiagonal form as returned by subroutine BD_CMP2 in its argument MAT.

The shape of MAT must verify: size(MAT, 1) >= size(MAT, 2) = n .

NSING (OUTPUT) integer(i4b) On output, NSING specifies the number of singular values which have been computed. Note that NSING may be greater than the optional argument LS, if multiple singular values at index LS make unique selection impossible.

If none of the optional arguments LS and THETA are used, NSING is set to size(MAT,2) and all the singular values are computed.

S (OUTPUT) real(stnd), dimension(:) On exit, S(1:NSING) contains the first NSING singular values of MAT. The other values in S (S(NSING+1:)) are flagged by a quiet NAN.

The size of S must be equal to size(MAT, 2) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit and the bisection or dqds algorithm converged for all the computed singular values to the desired accuracy ;
- FAILURE = true : indicates that some or all of the singular values failed to converge or were not computed. This is generally caused by unexpectedly inaccurate arithmetic. The sign of the incorrect singular values is set to negative.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MUL_SIZE (INPUT, OPTIONAL) integer(i4b) Internal parameter. MUL_SIZE must verify: 1 <= MUL_SIZE <= n, otherwise a default value is used. For better performance, at the expense of more workspace, a large value can be used.

The default is min(32, n).

VECTOR (INPUT, OPTIONAL) logical(lgl) On entry, if VECTOR is set to TRUE, a vectorized version of the bisection algorithm is used to compute the singular values SIGMA of the bidiagonal matrix BD.

This argument has no effect if DQDS is equal to true.

The default is VECTOR = false.

ABSTOL (INPUT, OPTIONAL) real(stnd) On entry, the absolute tolerance for the singular values. A singular value (or cluster) is considered to be located in the bisection algorithm if it been determined to lie in an interval whose width is ABSTOL or less.

Singular values will be computed most accurately when ABSTOL is set to the square root of the underflow threshold (e.g., sqrt(safmin)=sqrt(LAMCH('S')), not zero.

If ABSTOL is less than or equal to zero, or is not specified, then ULP * | BD' * BD | will be used, where | BD' * BD | means the 1-norm of the tridiagonal matrix BD' * BD (BD' means the transpose of BD) and ULP is the machine precision (distance from 1 to the next larger floating point number).

This argument has no effect if DQDS is equal to true.

LS (INPUT, OPTIONAL) integer(i4b) On entry, LS specifies the number of singular values which must be computed by the subroutine. On output, NSING may be different than LS if multiple singular values at index LS make unique selection impossible.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

LS must be greater than 0 and less or equal to size(MAT, 2) = n .

This argument has no effect if DQDS is equal to true.

The default is LS = size(MAT, 2) = n .

THETA (INPUT, OPTIONAL) real(stnd) On entry, THETA specifies that the singular values which are greater or equal to THETA must be computed. If none of the singular values are greater or equal to THETA, NSING is set to zero and S(:) to a quiet NAN.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

This argument has no effect if DQDS is equal to true.

The default is THETA = 0.

D (OUTPUT, OPTIONAL) real(stnd), dimension(:) The diagonal elements of the intermediate bidiagonal matrix BD

The size of D must be equal to size(MAT, 2) = n .

E (OUTPUT, OPTIONAL) real(stnd), dimension(:) The off-diagonal elements of the intermediate upper bidiagonal matrix BD:

E(i) = B(i-1,i) for i = 2,3,. . . ,n;

E(1) is arbitrary.

The size of E must be equal to size(MAT, 2) = n .

P (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, P is overwritten with the n-by-n orthogonal matrix P (stored column-wise or in packed form), the orthogonal matrix used to reduce MAT to bidiagonal form as returned by subroutine BD_CMP2 in its argument P.

The shape of P must verify: size(P, 1) = size(P, 2) = n .

GEN_P (INPUT, OPTIONAL) logical(lgl) On entry, this optional argument has an effect only if the optional argument P is also used.

In this case, if the optional argument GEN_P is used and is set to true, the orthogonal matrix P used to reduce MAT to bidiagonal form is generated on output of the subroutine in its argument P..

If GEN_P is set to false, the orthogonal matrix P is stored in factored form as products of elementary reflectors in the lower triangle of the array P.

See the description of BD_CMP2 subroutine for more details.

The default is true.

REORTHO (INPUT, OPTIONAL) logical(lgl) If the optional argument REORTHO is set to true, reorthogonalization of the matrix Q, used to reduce MAT to bidiagonal form, is performed (when needed). If this optional logical argument is set to false, reorthogonalization of Q is never performed and the columns of the matrix Q can be far from orthogonal to each other for input matrices MAT with a large condition number.

See the description of BD_CMP2 subroutine for more details.

The default is REORTHO = false.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the bidiagonal matrix BD is scaled before computing the singular values.

The default is to scale the bidiagonal matrix.

INIT (INPUT, OPTIONAL) logical(lgl) On entry, if INIT=true the initial intervals for the bisection steps are computed from estimates of the eigenvalues of the associated BD' * BD tridiagonal matrix obtained from the Pal-Walker-Kahan algorithm.

This argument has no effect if DQDS is equal to true.

The default is not to use the Pal-Walker-Kahan algorithm.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed.

If DQDS is set to true, all singular values are computed with the dqds algorithm The dqds algorithm is much faster than bisection when all singular values are requested in many cases and delivers about the same accuracy than bisection.

If DQDS is set to false, the requested singular values are computed by bisection.

The default is to use the dqds algorithm if all the singular values are requested and to use bisection if selected singular values are requested (e.g., if the optional arguments LS or THETA are present).

FAILURE_BD (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE_BD = false : indicates that maximum accuracy was obtained in the Ralha-Barlow one-sided bidiagonalization of MAT.
- FAILURE_BD = true : indicates that MAT is nearly singular and some loss of orthogonality can be expected in the Ralha-Barlow bidiagonalization algorithm.

See description of the BD_CMP2 subroutine for further details.

Further Details

The matrices Q, P and BD are computed with the help of the Ralha-Barlow one-sided method. Q is computed by a recurrence relationship and the first n columns of Q are stored in the argument MAT on exit. P is computed as a product of n-1 elementary reflectors (e.g. Householder transformations):

P = G(1) * G(2) * ... * G(n-1)

Each G(i) has the form:

G(i) = I + taup * v * v'

where taup is a real scalar, and v is a real vector. IF GEN_P is used and set to false, the n-1 G(i) elementary reflectors are stored in the lower triangle of the array P.

For the G(i) reflector, taup is stored in P(i+1,1) and v is stored in P(i+1:n,i+1). In addition, P(1,1) is set to -1 if GEN_P=false and is equal to 1 if GEN_P=true.

In other words, the value of P(1,1) indicates if the orthogonal matrix P is stored in factored form or not. Note that if n is equal to 1, elementary reflectors are not needed and consequently P(1,1) is set to 1, independently of the value of GEN_P.

This is the blocked version of the Ralha-Barlow one-sided algorithm. See the references (1), (2) and (3) for further details. Furthermore, the algorithm is parallelized if OPENMP is used.

Since Q is computed by a recurrence relationship, a loss of orthogonality of Q can be observed when the rectangular matrix MAT is singular or nearly singular or has a large condition number, see the reference (2) for details.

To correct partly this deficiency, partial reorthogonalization can be performed to ensure orthogonality at the expense of speed of computation. The reorthogonalization uses the Gram-Schmidt method described in the reference (4) and is activated if the optional logical argument REORTHO is set to true.

The reference (2) also explains how to handle the case of an exactly singular matrix MAT (a very rare event). However, in this subroutine, the partial reorthogonalization described in the reference (4), which is used when REORTHO is set true, corrects automatically this problem.

Now, let SIGMA(i), i=1,. . . ,n, be the singular values of the intermediate bidiagonal matrix BD in decreasing order of magnitude.

If DQDS is equal to false, the subroutine computes the LS largest singular values (or the singular values which are greater or equal to THETA) of BD by a bisection method (see the reference (5) below, Sec.8.5). The bisection method is applied (implicitly) to the associated n-by-n symmetric tridiagonal matrix BD' * BD whose eigenvalues are the squares of the singular values of BD by using the differential stationary form of the qd algorithm of Rutishauser (see the reference (6) below, Sec.3.1).

On the other hand, if DQDS is equal to true, all the singular values of BD are computed by the dqds algorithm (see the references (7) and (8).

If bisection is used, SELECT_SINGVAL_CMP4 subroutine is less accurate, but faster, than SELECT_SINGVAL_CMP3 subroutine since SELECT_SINGVAL_CMP3 works on the 2.n-by-2.n symmetric tridiagonal GOLUB-KAHAN form of BD, while SELECT_SINGVAL_CMP4 works implicitly on the associated n-by-n symmetric tridiagonal matrix BD' * BD whose eigenvalues are the squares of the singular values of BD.

For further details, see:

(1) **Ralha, R.M.S., 2003:** One-sided reduction to bidiagonal form. Linear Algebra Appl., No 358, pp. 219-238.

(2) **Barlow, J.L., Bosner, N., and Drmac, Z., 2005:** A new stable bidiagonal reduction algorithm. Linear Algebra Appl., No 397, pp. 35-84.

(3) **Bosner, N., and Barlow, J.L., 2007:** Block and Parallel versions of one-sided bidiagonalization. SIAM J. Matrix Anal. Appl., Volume 29, No 3, pp. 927-953.

(4) **Stewart, G.W., 2007:** Block Gram-Schmidt Orthogonalization. Report TR-4823, Department of Computer Science, College Park, University of Maryland.

(5) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(6) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

(7) **Fernando, K.V., and Parlett, B.N., 1994:** Accurate singular values and differential qd algorithms. Numer. Math., Volume 67, No 2, pp. 191-229.

(8) **Li, S., Gu, M., and Parlett, B. N., 2014:** An Improved DQDS Algorithm. SIAM Journal on Scientific Computing, Volume 36, No 3, C290-C308.

6.19.34 subroutine select_singval_cmp4 (mat, rmat, nsing, s, failure, sort, mul_size, vector, abstol, ls, theta, d, e, tauo, p, gen_p, reortho, scaling, init, dqds, failure_bd)

Purpose

SELECT_SINGVAL_CMP4 computes all or some of the greatest singular values of a real m-by-n matrix MAT with m>=n.

The Singular Value Decomposition (SVD) is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative.

The original matrix MAT is first reduced to upper bidiagonal form by a two-step algorithm :

A QR factorization of the real m-by-n matrix MAT is first computed

MAT = O * R

where O is orthogonal and R is upper triangular.

In a second step, the n-by-n upper triangular matrix R is reduced to upper bidiagonal form BD by an orthogonal transformation :

Q' * R * P = BD

where Q and P are orthogonal and BD is an upper bidiagonal matrix (see the reference (5) below). The Ralha-Barlow one-sided method is used in this second step (see the references (1) to (3) below).

SELECT_SINGVAL_CMP4 computes O, BD, Q and P.

The singular values SIGMA of the bidiagonal matrix BD, which are also the singular values of MAT, are then computed by a bisection algorithm (see the reference (5) below, Sec.8.5). The bisection method is applied (implicitly) to the associated n-by-n symmetric tridiagonal matrix BD' * BD whose eigenvalues are the squares of the singular values of BD by using the differential stationary form of the qd algorithm of Rutishauser (see the reference (6) below, Sec.3.1). Alternatively, at the user option, all singular values can be computed by the differential quotient difference with shifts (dqds) algorithm (see the references (7) and (8)).

The routine outputs (parts of) SIGMA, and optionally O, Q and P and BD for a given matrix MAT. The matrix O is stored in factored form in the argument MAT if the optional argument TAUO is present or explicitly computed if this argument is absent. The first n columns of Q are stored in the argument RMAT. Finally, P is stored in the optional argument P. P is stored in factored form or explicitly generated depending on the value of the optional logical argument GEN_P.

SIGMA, O, Q, P and BD may then be used to obtain selected singular vectors of MAT with subroutines BD_INVITER2 or BD_DEFLATE2.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, the elements on and below the diagonal, with the array TAUO, represent the orthogonal matrix O of the QR factorization of MAT, as a product of elementary reflectors, if the argument TAUO is present. Otherwise, the argument MAT contains the first n columns of O (stored column-wise) on output.

See Further Details.

The shape of MAT must verify: size(MAT, 1) >= size(MAT, 2) = n .

RMAT (OUTPUT) real(stnd), dimension(:,:)

On exit, RMAT contains the first n columns of Q (stored column-wise), the orthogonal matrix used to reduce R to bidiagonal form as returned by subroutine BD_CMP2 in its argument MAT.

See Further Details.

The shape of RMAT must verify: size(RMAT, 1) = size(RMAT, 2) = size(MAT, 2) = n .

NSING (OUTPUT) integer(i4b) On output, NSING specifies the number of singular values which have been computed. Note that NSING may be greater than the optional argument LS, if multiple singular values at index LS make unique selection impossible.

If none of the optional arguments LS and THETA are used, NSING is set to size(MAT,2) and all the singular values are computed.

S (OUTPUT) real(stnd), dimension(:) On exit, S(1:NSING) contains the first NSING singular values of MAT. The other values in S (S(NSING+1:)) are flagged by a quiet NAN.

The size of S must be equal to size(MAT, 2) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit and the bisection or dqds algorithm converged for all the computed singular values to the desired accuracy ;
- FAILURE = true : indicates that some or all of the singular values failed to converge or were not computed. This is generally caused by unexpectedly inaccurate arithmetic. The sign of the incorrect singular values is set to negative.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

MUL_SIZE (INPUT, OPTIONAL) integer(i4b) Internal parameter. MUL_SIZE must verify: 1 <= MUL_SIZE <= n, otherwise a default value is used. For better performance, at the expense of more workspace, a large value can be used.

The default is min(32, n).

VECTOR (INPUT, OPTIONAL) logical(lgl) On entry, if VECTOR is set to TRUE, a vectorized version of the bisection algorithm is used to compute the singular values SIGMA of the bidiagonal matrix BD.

This argument has no effect if DQDS is equal to true.

The default is VECTOR = false.

ABSTOL (INPUT, OPTIONAL) real(stnd) On entry, the absolute tolerance for the singular values. A singular value (or cluster) is considered to be located in the bisection algorithm if it been determined to lie in an interval whose width is ABSTOL or less.

Singular values will be computed most accurately when ABSTOL is set to the square root of the underflow threshold (e.g., sqrt(safmin)=sqrt(LAMCH('S')), not zero.

If ABSTOL is less than or equal to zero, or is not specified, then ULP * | BD' * BD | will be used, where | BD' * BD | means the 1-norm of the tridiagonal matrix BD' * BD (BD' means the transpose of BD) and ULP is the machine precision (distance from 1 to the next larger floating point number).

This argument has no effect if DQDS is equal to true.

LS (INPUT, OPTIONAL) integer(i4b) On entry, LS specifies the number of singular values which must be computed by the subroutine. On output, NSING may be different than LS if multiple singular values at index LS make unique selection impossible.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

LS must be greater than 0 and less or equal to size(MAT, 2) = n .

This argument has no effect if DQDS is equal to true.

The default is LS = size(MAT, 2) = n .

THETA (INPUT, OPTIONAL) real(stnd) On entry, THETA specifies that the singular values which are greater or equal to THETA must be computed. If none of the singular values are greater or equal to THETA, NSING is set to zero and S(:) to a quiet NAN.

Only one of the optional arguments LS and THETA must be specified, otherwise the subroutine will stop with an error message.

This argument has no effect if DQDS is equal to true.

The default is THETA = 0.

D (OUTPUT, OPTIONAL) real(stnd), dimension(:) The diagonal elements of the intermediate bidiagonal matrix BD

The size of D must be equal to size(MAT, 2) = n .

E (OUTPUT, OPTIONAL) real(stnd), dimension(:) The off-diagonal elements of the intermediate upper bidiagonal matrix BD:

E(i) = B(i-1,i) for i = 2,3,. . . ,n;

E(1) is arbitrary.

The size of E must be equal to size(MAT, 2) = n .

TAUO (OUTPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix O of the QR decomposition of MAT.

If the optional argument TAUO is present, the orthogonal matrix O is stored in factored form, as a product of elementary reflectors, in the argument MAT on exit.

If the optional argument TAUO is absent, the first n columns of the orthogonal matrix O are explicitly generated and stored in the argument MAT on exit.

See description of the argument MAT above and Further Details below.

The size of TAUO must be equal to size(MAT, 2) = n .

P (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, P is overwritten with the n-by-n orthogonal matrix P (stored column-wise or in packed form), the orthogonal matrix used to reduce MAT to bidiagonal form as returned by subroutine BD_CMP2 in its argument P.

The shape of P must verify: size(P, 1) = size(P, 2) = n .

GEN_P (INPUT, OPTIONAL) logical(lgl) On entry, this optional argument has an effect only if the optional argument P is also used.

In this case, if the optional argument GEN_P is used and is set to true, the orthogonal matrix P used to reduce MAT to bidiagonal form is generated on output of the subroutine in its argument P..

If GEN_P is set to false, the orthogonal matrix P is stored in factored form as products of elementary reflectors in the lower triangle of the array P.

See the description of BD_CMP2 subroutine for more details.

The default is true.

REORTHO (INPUT, OPTIONAL) logical(lgl) If the optional argument REORTHO is set to true, reorthogonalization of the matrix Q, used to reduce MAT to bidiagonal form, is performed (when needed). If this optional logical argument is set to false, reorthogonalization of Q is never performed and the columns of the matrix Q can be far from orthogonal to each other for input matrices MAT with a large condition number.

See the description of BD_CMP2 subroutine for more details.

The default is REORTHO = false.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the bidiagonal matrix BD is scaled before computing the singular values.

The default is to scale the bidiagonal matrix.

INIT (INPUT, OPTIONAL) logical(lgl) On entry, if INIT=true the initial intervals for the bisection steps are computed from estimates of the eigenvalues of the associated BD' * BD tridiagonal matrix obtained from the Pal-Walker-Kahan algorithm.

This argument has no effect if DQDS is equal to true.

The default is not to use the Pal-Walker-Kahan algorithm.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed.

If DQDS is set to true, all singular values are computed with the dqds algorithm The dqds algorithm is much faster than bisection when all singular values are requested in many cases and delivers about the same accuracy than bisection.

If DQDS is set to false, the requested singular values are computed by bisection.

The default is to use the dqds algorithm if all the singular values are requested and to use bisection if selected singular values are requested (e.g., if the optional arguments LS or THETA are present).

FAILURE_BD (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE_BD = false : indicates that maximum accuracy was obtained in the Ralha-Barlow one-sided bidiagonalization of MAT.
- FAILURE_BD = true : indicates that MAT is nearly singular and some loss of orthogonality can be expected in the Ralha-Barlow bidiagonalization algorithm.

See description of the BD_CMP2 subroutine for further details.

Further Details

the matrix O of the QR factorization of MAT is represented as a product of elementary reflectors

O = W(1) * W(2) * ... * W(n)

Each W(i) has the form

W(i) = I + tauo * (v * v') ,

where tauo is a real scalar and v is a real m-element vector with v(1:i-1) = 0. v(i:m) is stored on exit in MAT(i:m,i) and tauo in TAUO(i). If the optional argument TAUO is absent, the first n columns of O are generated and stored in the argument MAT.

The matrix O is stored in factored form if the optional argument TAUO is present or explicitly computed if this argument is absent.

A blocked algorithm is used for computing the QR factorization of MAT. Furthermore, the computations are parallelized if OPENMP is used.

After, the initial QR factorization of MAT, the upper triangular matrix R is reduced to upper bidiagonal form BD:

Q' * R * P = BD

The matrices Q, P and BD are computed with the help of the Ralha-Barlow one-sided method. Q is computed by a recurrence relationship and the first n columns of Q are stored in the argument RMAT on exit. P is computed as a product of n-1 elementary reflectors (e.g. Householder transformations):

P = G(1) * G(2) * ... * G(n-1)

Each G(i) has the form:

G(i) = I + taup * v * v'

where taup is a real scalar, and v is a real vector. IF GEN_P is used and set to false, the n-1 G(i) elementary reflectors are stored in the lower triangle of the array P.

For the G(i) reflector, taup is stored in P(i+1,1) and v is stored in P(i+1:n,i+1). In addition, P(1,1) is set to -1 if GEN_P=false and is equal to 1 if GEN_P=true.

In other words, the value of P(1,1) indicates if the orthogonal matrix P is stored in factored form or not. Note that if n is equal to 1, elementary reflectors are not needed and consequently P(1,1) is set to 1, independently of the value of GEN_P.

This is the blocked version of the Ralha-Barlow one-sided algorithm. See the references (1), (2) and (3) for further details. Furthermore, the algorithm is parallelized if OPENMP is used.

Since Q is computed by a recurrence relationship, a loss of orthogonality of Q can be observed when the rectangular matrix MAT is singular or nearly singular or has a large condition number, see the reference (2) for details.

To correct partly this deficiency, partial reorthogonalization can be performed to ensure orthogonality at the expense of speed of computation. The reorthogonalization uses the Gram-Schmidt method described in the reference (4) and is activated if the optional logical argument REORTHO is set to true.

The reference (2) also explains how to handle the case of an exactly singular matrix MAT (a very rare event). However, in this subroutine, the partial reorthogonalization described in the reference (4), which is used when REORTHO is set true, corrects automatically this problem.

Now, let SIGMA(i), i=1,...,n, be the singular values of the intermediate bidiagonal matrix BD in decreasing order of magnitude.

If DQDS is equal to false, the subroutine computes the LS largest singular values (or the singular values which are greater or equal to THETA) of BD by a bisection method (see the reference (5) below, Sec.8.5). The bisection method is applied (implicitly) to the associated n-by-n symmetric tridiagonal matrix BD' * BD whose eigenvalues are the squares of the singular values of BD by using the differential stationary form of the qd algorithm of Rutishauser (see the reference (6) below, Sec.3.1).

On the other hand, if DQDS is equal to true, all the singular values of BD are computed by the dqds algorithm (see the references (7) and (8).

If bisection is used, SELECT_SINGVAL_CMP4 subroutine is less accurate, but faster, than SELECT_SINGVAL_CMP3 subroutine since SELECT_SINGVAL_CMP3 works on the 2n-by-2n symmetric tridiagonal GOLUB-KAHAN form of BD, while SELECT_SINGVAL_CMP4 works implicitly on the associated n-by-n symmetric tridiagonal matrix BD' * BD whose eigenvalues are the squares of the singular values of BD.

For further details, see:

(1) **Ralha, R.M.S., 2003:** One-sided reduction to bidiagonal form. Linear Algebra Appl., No 358, pp. 219-238.

(2) **Barlow, J.L., Bosner, N., and Drmac, Z., 2005:** A new stable bidiagonal reduction algorithm. Linear Algebra Appl., No 397, pp. 35-84.

(3) **Bosner, N., and Barlow, J.L., 2007:** Block and Parallel versions of one-sided bidiagonalization. SIAM J. Matrix Anal. Appl., Volume 29, No 3, pp. 927-953.

(4) **Stewart, G.W., 2007:** Block Gram-Schmidt Orthogonalization. Report TR-4823, Department of Computer Science, College Park, University of Maryland.

(5) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(6) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

(7) **Fernando, K.V., and Parlett, B.N., 1994:** Accurate singular values and differential qd algorithms. Numer. Math., Volume 67, No 2, pp. 191-229.

(8) **Li, S., Gu, M., and Parlett, B. N., 2014:** An Improved DQDS Algorithm. SIAM Journal on Scientific Computing, Volume 36, No 3, C290-C308.

6.19.35 subroutine svd_cmp (mat, s, failure, v, sort, mul_size, maxiter, max_francis_steps, perfect_shift, bisect, dqds, use_svd2)

Purpose

SVD_CMP computes the Singular Value Decomposition (SVD) of a real m-by-n matrix MAT. The SVD is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative. The columns of U and V are the left and right singular vectors of MAT.

SVD_CMP computes only the first min(m,n) columns of U and V (e.g. the left and right singular vectors of MAT in the thin SVD of MAT).

The routine returns the first min(m,n) singular values and the associated left and right singular vectors.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is overwritten with the first min(m,n) columns of U, the left singular vectors.

S (OUTPUT) real(stnd), dimension(:) The singular values of MAT.

The size of S must verify: size(S) = min(m,n) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the bidiagonal SVD of an intermediate bidiagonal form B of MAT.

V (OUTPUT) real(stnd), dimension(:,:) On exit, V contains the first min(m,n) columns of V, the right singular vectors.

The shape of V must verify:

- size(V, 1) = n,
- size(V, 2) = min(m,n).

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The singular vectors are rearranged accordingly.

MUL_SIZE (INPUT, OPTIONAL) integer(i4b) Internal parameter. MUL_SIZE must verify: 1 <= MUL_SIZE <= max(m,n), otherwise a default value is used. MUL_SIZE can be increased or decreased to improve the performance of the algorithm.

The default value is 32.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm.

The bidiagonal SVD algorithm of an intermediate bidiagonal form B of MAT fails to converge if the number of QR sweeps exceeds MAXITER * min(m,n). Convergence usually occurs in about 2 * min(m,n) QR sweeps.

The default is 10.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm.

MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of min(m,n) and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

USE_SVD2 (INPUT, OPTIONAL) logical(lgl) If the optional argument USE_SVD2 is used and is set to true, an alternate SVD algorithm which used less workspace (but which may be slower) is automatically used if m is much larger than n or if n is much larger than m (e.g. if max(m,n)>=1.5 * min(m,n)).

Further Details

Computing the SVD of a rectangular matrix in subroutine SVD_CMP consists of three steps:

1) reduction of the rectangular matrix to bidiagonal form via orthogonal transformations (e.g. Householder transformations);

2) in place accumulation of the orthogonal transformations used in the reduction to bidiagonal form;

3) computation of the SVD of the bidiagonal matrix.

For further details, on the SVD of a rectangular matrix and the algorithm to compute it, see the references (1) or (2).

All the three steps of the SVD algorithm (e.g. the reduction to bidiagonal form, accumulation of the Householder transformations used in the reduction to bidiagonal form and computation of the SVD of the bidiagonal matrix) are parallelized if OPENMP is used.

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

6.19.36 subroutine svd_cmp (mat, s, failure, sort, mul_size, maxiter, bisect, dqds, d, e, tauq, taup)

Purpose

SVD_CMP computes the singular values of a real m-by-n matrix MAT.

The Singular Value Decomposition (SVD) is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative.

The original matrix MAT is first reduced to upper or lower bidiagonal form BD by an orthogonal transformation:

Q' * MAT * P = BD

where Q and P are orthogonal. The singular values SIGMA of the bidiagonal matrix BD are then computed by the bidiagonal implicit QR algorithm (if BISECT=false and DQDS=false), the bisection method (if BISECT=true) or the dqds algorithm (if DQDS=true) applied to the bidiagonal matrix BD.

The routine outputs SIGMA and optionally Q and P (in packed form), and BD for a given matrix MAT. SIGMA, Q, P and BD may then be used to obtain selected singular vectors with subroutines BD_INVITER, BD_INVITER2, BD_DEFLATE or BD_DEFLATE2.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is destroyed and if TAUQ or TAUP are present MAT is overwritten as follows:

- if m >= n, the elements on and below the diagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array TAUP, represent the orthogonal matrix P as a product of elementary reflectors;
- if m < n, the elements below the diagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements on and above the diagonal, with the array TAUP, represent the orthogonal matrix P as a product of elementary reflectors.

See Further Details.

S (OUTPUT) real(stnd), dimension(:) On exit, the singular values SIGMA of MAT if FAILURE is equal to false. If FAILURE is equal to true, the sign of the incorrect singular values is set to negative.

The size of S must be min(size(MAT,1) , size(MAT,2)) = min(m,n).

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the implicit QR, bisection or dqds algorithm used to compute the singular values of the bidiagonal form BD of the input m-by-n matrix MAT did not converge and that full accuracy was not attained in the bidiagonal SVD of this intermediate bidiagonal form BD of MAT. The sign of the incorrect singular values is set to negative.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

By default, the singular values are not sorted.

MUL_SIZE (INPUT, OPTIONAL) integer(i4b) Internal parameter. MUL_SIZE must verify: 1 <= MUL_SIZE <= max(m,n), otherwise a default value is used. For better performance, at the expense of more workspace, a large value can be used.

The default is 32.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal implicit QR phase of the SVD algorithm.

The bidiagonal SVD algorithm of an intermediate bidiagonal form BD of MAT fails to converge if the number of QR sweeps exceeds MAXITER * min(m,n). Convergence usually occurs in about 2 * min(m,n) QR sweeps.

This argument has no effect if BISECT or DQDS is equal to true.

The default is 10.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values of the intermediate min(m,n)-by-min(m,n) bidiagonal matrix BD are computed.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slower execution time.

If BISECT is set to false, singular values are computed with the bidiagonal implicit QR or dqds algorithm applied to the associated min(m,n)-by-min(m,n) bidiagonal matrix BD.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values of the intermediate min(m,n)-by-min(m,n) bidiagonal matrix BD are computed.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition. Moreover, the dqds algorithm is usually also faster than the bisection or bidiagonal implicit QR algorithms and is the default method used in SVD_CMP for computing singular values of the bidiagonal form BD of the input matrix MAT.

If DQDS is set to false, singular values are computed with the bidiagonal implicit QR or bisection algorithm applied to the associated min(m,n)-by-min(m,n) bidiagonal matrix BD.

If both optional arguments BISECT and DQDS are specified with the value true, the dqds algorithm is used.

If both optional arguments BISECT and DQDS are specified with the value false, the bidiagonal implicit QR algorithm is used.

The default is true.

D (OUTPUT, OPTIONAL) real(stnd), dimension(:) The diagonal elements of the intermediate bidiagonal matrix BD

The size of D must be min(size(MAT,1) , size(MAT,2)) = min(m,n).

E (OUTPUT, OPTIONAL) real(stnd), dimension(:) The off-diagonal elements of the intermediate bidiagonal matrix BD:

- if m >= n, E(i) = BD(i-1,i) for i = 2,3,. . . ,n;
- if m < n, E(i) = BD(i,i-1) for i = 2,3,. . . ,m.

The size of E must be min(size(MAT,1) , size(MAT,2)) = min(m,n).

TAUQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix Q. See Further Details.

The size of TAUQ must be min(size(MAT,1) , size(MAT,2)) = min(m,n).

TAUP (OUTPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix P. See Further Details.

The size of TAUP must be min(size(MAT,1) , size(MAT,2)) = min(m,n).

Further Details

Computing the singular values of a rectangular matrix in subroutine SVD_CMP consists of two steps:

1) reduction of the rectangular matrix to bidiagonal form BD, see the references (1) and (2);

2) computation of the singular values of the min(m,n)-by-min(m,n) bidiagonal matrix BD by a bidiagonal implicit QR algorithm (if BISECT=false and DQDS=false), a dqds algorithm (if DQDS=true) or a bisection algorithm (if BISECT=true). See the references (1), (2), (3) and (4) for a description of these different algorithms.

Note that if max(m,n) is much larger than min(m,n) and the optional arguments TAUQ and TAUP are not used, the rectangular matrix is first reduced to upper or lower triangular form by a QR or LQ factorization and the reduction algorithm is applied to the resulting triangular factor. The singular values of the rectangular matrix are then obtained from those of the triangular factor. This usually gives a large speedup in the computations.

The matrices Q and P in the bidiagonal reduction of the input m-by-n matrix MAT are represented as products of elementary reflectors:

- If m >= n,

  Q = H(1) * H(2) * . . . * H(n) and P = G(1) * G(2) * . . . * G(n-1)

  Each H(i) and G(i) has the form:

  H(i) = I + tauq * u * u' and G(i) = I + taup * v * v'

  where tauq and taup are real scalars, and u and v are real vectors. Moreover, u(1:i-1) = 0 and v(1:i) = 0.

  If TAUQ or TAUP are present :

  – u(i:m) is stored on exit in MAT(i:m,i);

  – v(i+1:n) is stored on exit in MAT(i,i+1:n).

  If TAUQ is present : tauq is stored in TAUQ(i).

  If TAUP is present : taup is stored in TAUP(i).

- If m < n,

  Q = H(1) * H(2) * . . . * H(m-1) and P = G(1) * G(2) * . . . * G(m)

  Each H(i) and G(i) has the form:

  H(i) = I + tauq * u * u' and G(i) = I + taup * v * v'

  where tauq and taup are real scalars, and u and v are real vectors. Moreover, u(1:i) = 0 and v(1:i-1) = 0.

  If TAUQ or TAUP are present :

  – u(i+1:m) is stored on exit in MAT(i+1:m,i);

  – v(i:n) is stored on exit in MAT(i,i:n).

  If TAUQ is present : tauq is stored in TAUQ(i).

  If TAUP is present : taup is stored in TAUP(i).

The contents of MAT on exit, if TAUQ or TAUP are present, are illustrated by the following examples:

- m = 6 and n = 5 (m >= n):

(u1 v1 v1 v1 v1)

(u1 u2 v2 v2 v2)

(u1 u2 u3 v3 v3)

(u1 u2 u3 u4 v4)

(u1 u2 u3 u4 u5)

(u1 u2 u3 u4 u5)

- m = 5 and n = 6 (m < n):

  (v1 v1 v1 v1 v1 v1)

  (u1 v2 v2 v2 v2 v2)

  (u1 u2 v3 v3 v3 v3)

  (u1 u2 u3 v4 v4 v4)

  (u1 u2 u3 u4 v5 v5)

where ui denotes an element of the vector defining H(i), and vi an element of the vector defining G(i).

For further details, on the SVD of a rectangular matrix and the algorithms to compute it, see the references (1) or (2). In SVD_CMP subroutine, the reduction to bidiagonal form by orthogonal transformations is parallelized if OPENMP is used, the computation of the singular values is also parallelized if OPENMP is used and BISECT is used with the value true. Further details on the dqds algorithm are given in the references (3) and (4).

For more informations, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Fernando, K.V., and Parlett, B.N., 1994:** Accurate singular values and differential qd algorithms. Numer. Math., Volume 67, No 2, pp. 191-229.

(4) **Li, S., Gu, M., and Parlett, B. N., 2014:** An Improved DQDS Algorithm. SIAM Journal on Scientific Computing, Volume 36, No 3, C290-C308.

6.19.37 subroutine svd_cmp2 (mat, s, failure, u_vt, sort, mul_size, maxiter, max_francis_steps, perfect_shift, bisect, dqds, use_svd2)

Purpose

SVD_CMP2 computes the Singular Value Decomposition (SVD) of a real m-by-n matrix MAT. The SVD is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative. The columns of U and V are the left and right singular vectors of MAT.

SVD_CMP2 computes only the first min(m,n) columns of U and V (e.g. the left and right singular vectors of MAT in the thin SVD of MAT).

The routine returns the first min(m,n) singular values and the associated left and right singular vectors. The right singular vectors are returned row-wise.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit:

- if m>=n, MAT is overwritten with the first min(m,n) columns of U (the left singular vectors, stored column-wise);
- if m<n, MAT is overwritten with the first min(m,n) rows of V' (the right singular vectors, stored row-wise).

S (OUTPUT) real(stnd), dimension(:) The singular values of MAT.

The size of S must verify: size(S) = min(m,n) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the bidiagonal SVD of an intermediate bidiagonal form B of MAT.

U_VT (OUTPUT) real(stnd), dimension(:,:) On exit:

- if m>=n, U_VT contains the n-by-n orthogonal matrix V';
- if m<n, U_VT contains the m-by-m orthogonal matrix U.

The shape of U_VT must verify: size(U_VT, 1) = size(U_VT, 2) = min(m,n).

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The singular vectors are rearranged accordingly.

MUL_SIZE (INPUT, OPTIONAL) integer(i4b) Internal parameter. MUL_SIZE must verify 1 <= MUL_SIZE <= max(m,n), otherwise a default value is used. MUL_SIZE can be increased or decreased to improve the performance of the algorithm.

The default value is 32.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm.

The bidiagonal SVD algorithm of an intermediate bidiagonal form B of MAT fails to converge if the number of QR sweeps exceeds MAXITER * min(m,n). Convergence usually occurs in about 2 * min(m,n) QR sweeps.

The default is 10.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm.

MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of min(m,n) and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

USE_SVD2 (INPUT, OPTIONAL) logical(lgl) If the optional argument USE_SVD2 is used and is set to true, an alternate SVD algorithm which used less workspace (but which may be slower) is automatically used if m is much larger than n or if n is much larger than m (e.g. if max(m,n)>=1.5 * min(m,n)).

Further Details

Computing the SVD of a rectangular matrix in subroutine SVD_CMP2 consists of three steps:

1) reduction of the rectangular matrix to bidiagonal form via orthogonal transformations (e.g. Householder transformations);

2) in place accumulation of the orthogonal transformations used in the reduction to bidiagonal form;

3) computation of the SVD of the bidiagonal matrix.

For further details, on the SVD of a rectangular matrix and the algorithm to compute it, see the references (1) or (2).

All the three steps of the SVD algorithm (e.g. the reduction to bidiagonal form, accumulation of the Householder transformations used in the reduction to bidiagonal form and computation of the SVD of the bidiagonal matrix) are parallelized if OPENMP is used.

For more informations, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

6.19.38 subroutine svd_cmp2 (mat, s, failure, sort, mul_size, maxiter, bisect, dqds, d, e, tauq, taup)

Purpose

SVD_CMP2 computes the singular values of a real m-by-n matrix MAT.

The Singular Value Decomposition (SVD) is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative.

The original matrix MAT is first reduced to upper or lower bidiagonal form BD by an orthogonal transformation:

Q' * MAT * P = BD

where Q and P are orthogonal. The singular values SIGMA of the bidiagonal matrix BD are then computed by the bidiagonal implicit QR algorithm (if BISECT=false and DQDS=false), the bisection method (if BISECT=true) or the dqds algorithm (if DQDS=true) applied to the bidiagonal matrix BD.

The routine outputs SIGMA and optionally Q and P (in packed form), and BD for a given matrix MAT. SIGMA, Q, P and BD may then be used to obtain selected singular vectors with subroutines BD_INVITER, BD_INVITER2, BD_DEFLATE or BD_DEFLATE2.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is destroyed and if TAUQ or TAUP are present MAT is overwritten as follows:

- if m >= n, the elements on and below the diagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array TAUP, represent the orthogonal matrix P as a product of elementary reflectors;
- if m < n, the elements below the diagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements on and above the diagonal, with the array TAUP, represent the orthogonal matrix P as a product of elementary reflectors.

See Further Details.

S (OUTPUT) real(stnd), dimension(:) On exit, the singular values SIGMA of MAT if FAILURE is equal to false. If FAILURE is equal to true, the sign of the incorrect singular values is set to negative.

The size of S must be min(size(MAT,1) , size(MAT,2)) = min(m,n).

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the implicit QR, bisection or dqds algorithm used to compute the singular values of the bidiagonal form BD of the input m-by-n matrix MAT did not converge and that full accuracy was not attained in the bidiagonal SVD of this intermediate bidiagonal form BD of MAT. The sign of the incorrect singular values is set to negative.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

By default, the singular values are not sorted.

MUL_SIZE (INPUT, OPTIONAL) integer(i4b) Internal parameter. MUL_SIZE must verify: 1 <= MUL_SIZE <= max(m,n), otherwise a default value is used. For better performance, at the expense of more workspace, a large value can be used.

The default is 32.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal implicit QR phase of the SVD algorithm.

The bidiagonal SVD algorithm of an intermediate bidiagonal form BD of MAT fails to converge if the number of QR sweeps exceeds MAXITER * min(m,n). Convergence usually occurs in about 2 * min(m,n) QR sweeps.

This argument has no effect if BISECT or DQDS is equal to true.

The default is 10.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values of the intermediate min(m,n)-by-min(m,n) bidiagonal matrix BD are computed.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slower execution time.

If BISECT is set to false, singular values are computed with the bidiagonal implicit QR or dqds algorithm applied to the associated min(m,n)-by-min(m,n) bidiagonal matrix BD.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values of the intermediate min(m,n)-by-min(m,n) bidiagonal matrix BD are computed.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition. Moreover, the dqds algorithm is usually also faster than the bisection or bidiagonal implicit QR algorithms and is the default method used in SVD_CMP2 for computing singular values of the bidiagonal form BD of the input matrix MAT.

If DQDS is set to false, singular values are computed with the bidiagonal implicit QR or bisection algorithm applied to the associated min(m,n)-by-min(m,n) bidiagonal matrix BD.

If both optional arguments BISECT and DQDS are specified with the value true, the dqds algorithm is used.

If both optional arguments BISECT and DQDS are specified with the value false, the bidiagonal implicit QR algorithm is used.

The default is true.

D (OUTPUT, OPTIONAL) real(stnd), dimension(:) The diagonal elements of the intermediate bidiagonal matrix BD

The size of D must be min(size(MAT,1) , size(MAT,2)) = min(m,n).

E (OUTPUT, OPTIONAL) real(stnd), dimension(:) The off-diagonal elements of the intermediate bidiagonal matrix BD:

- if m >= n, E(i) = BD(i-1,i) for i = 2,3,. . . ,n;
- if m < n, E(i) = BD(i,i-1) for i = 2,3,. . . ,m.

The size of E must be min(size(MAT,1) , size(MAT,2)) = min(m,n).

TAUQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix Q. See Further Details.

The size of TAUQ must be min(size(MAT,1) , size(MAT,2)) = min(m,n).

TAUP (OUTPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors which represent the orthogonal matrix P. See Further Details.

The size of TAUP must be min(size(MAT,1) , size(MAT,2)) = min(m,n).

Further Details

Computing the singular values of a rectangular matrix in subroutine SVD_CMP2 consists of two steps:

1) reduction of the rectangular matrix to bidiagonal form BD, see the references (1) and (2);

2) computation of the singular values of the min(m,n)-by-min(m,n) bidiagonal matrix BD by a bidiagonal implicit QR algorithm (if BISECT=false and DQDS=false), a dqds algorithm (if DQDS=true) or a bisection algorithm (if BISECT=true). See the references (1), (2), (3) and (4) for a description of these different algorithms.

Note that if max(m,n) is much larger than min(m,n) and the optional arguments TAUQ and TAUP are not used, the rectangular matrix is first reduced to upper or lower triangular form by a QR or LQ factorization and the reduction algorithm is applied to the resulting triangular factor. The singular values of the rectangular matrix are then obtained from those of the triangular factor. This usually gives a large speedup in the computations.

The matrices Q and P in the bidiagonal reduction of the input m-by-n matrix MAT are represented as products of elementary reflectors:

- If m >= n,

  Q = H(1) * H(2) * . . . * H(n) and P = G(1) * G(2) * . . . * G(n-1)

  Each H(i) and G(i) has the form:

  H(i) = I + tauq * u * u' and G(i) = I + taup * v * v'

  where tauq and taup are real scalars, and u and v are real vectors. Moreover, u(1:i-1) = 0 and v(1:i) = 0.

  If TAUQ or TAUP are present :

  – u(i:m) is stored on exit in MAT(i:m,i);

  – v(i+1:n) is stored on exit in MAT(i,i+1:n).

  If TAUQ is present : tauq is stored in TAUQ(i).

  If TAUP is present : taup is stored in TAUP(i).

- If m < n,

  Q = H(1) * H(2) * . . . * H(m-1) and P = G(1) * G(2) * . . . * G(m)

  Each H(i) and G(i) has the form:

  H(i) = I + tauq * u * u' and G(i) = I + taup * v * v'

  where tauq and taup are real scalars, and u and v are real vectors. Moreover, u(1:i) = 0 and v(1:i-1) = 0.

  If TAUQ or TAUP are present :

  – u(i+1:m) is stored on exit in MAT(i+1:m,i);

  – v(i:n) is stored on exit in MAT(i,i:n).

If TAUQ is present : tauq is stored in TAUQ(i).

If TAUP is present : taup is stored in TAUP(i).

The contents of MAT on exit, if TAUQ or TAUP are present, are illustrated by the following examples:

- m = 6 and n = 5 (m >= n):

  (u1 v1 v1 v1 v1)

  (u1 u2 v2 v2 v2)

  (u1 u2 u3 v3 v3)

  (u1 u2 u3 u4 v4)

  (u1 u2 u3 u4 u5)

  (u1 u2 u3 u4 u5)

- m = 5 and n = 6 (m < n):

  (v1 v1 v1 v1 v1 v1)

  (u1 v2 v2 v2 v2 v2)

  (u1 u2 v3 v3 v3 v3)

  (u1 u2 u3 v4 v4 v4)

  (u1 u2 u3 u4 v5 v5)

where ui denotes an element of the vector defining H(i), and vi an element of the vector defining G(i).

For further details, on the SVD of a rectangular matrix and the algorithms to compute it, see the references (1) or (2). In SVD_CMP2 subroutine, the reduction to bidiagonal form by orthogonal transformations is parallelized if OPENMP is used, the computation of the singular values is also parallelized if OPENMP is used and BISECT is used with the value true. Further details on the dqds algorithm are given in the references (3) and (4).

For more informations, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Fernando, K.V., and Parlett, B.N., 1994:** Accurate singular values and differential qd algorithms. Numer. Math., Volume 67, No 2, pp. 191-229.

(4) **Li, S., Gu, M., and Parlett, B. N., 2014:** An Improved DQDS Algorithm. SIAM Journal on Scientific Computing, Volume 36, No 3, C290-C308.

6.19.39 subroutine svd_cmp3 (mat, s, failure, u_v, sort, maxiter, max_francis_steps, perfect_shift, bisect, dqds, reortho, failure_bd)

Purpose

SVD_CMP3 computes the Singular Value Decomposition (SVD) of a real m-by-n matrix MAT. The SVD is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative. The columns of U and V are, respectively, the left and right singular vectors of MAT.

The routine returns the first min(m,n) singular values and the associated left and right singular vectors. The right singular vectors are returned row-wise if m<n.

MAT (or MAT' if m<n) is first reduced to bidiagonal form B with the help of the Ralha-Barlow one-sided bidiagonalization algorithm, see the references (1) and (2).

The singular values, left and right singular vectors of B are then computed by the bidiagonal implicit QR algorithm applied to B, see the references (3) and (4).

The singular vectors of MAT are finally computed by a back transformation algorithm.

In cases of a very large condition number of MAT, SVD_CMP3 may compute left (right if m<n) singular vectors of MAT, which are not numerically orthogonal (see Further Details). However, the largest left (right if m<n) singular vectors of MAT are always numerically orthogonal even if MAT is singular or nearly singular (see Further Details).

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit:

- if m>=n, MAT is overwritten with the first n columns of U (the left singular vectors, stored column-wise);
- if m<n, MAT is overwritten with the first m rows of V' (the first m right singular vectors, stored row-wise);

S (OUTPUT) real(stnd), dimension(:) The singular values of MAT.

The size of S must verify: size(S) = min(m,n) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the bidiagonal SVD algorithm did not converge and that full accuracy was not attained in the bidiagonal SVD of an intermediate bidiagonal form B of MAT.

U_V (OUTPUT) real(stnd), dimension(:,:) On exit:

- if m>=n, U_V contains the n-by-n orthogonal matrix V;
- if m<n, U_V contains the m-by-m orthogonal matrix U.

The shape of U_V must verify: size(U_V, 1) = size(U_V, 2) = min(m,n) .

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. If this argument is not used the singular values are not sorted. The singular vectors are rearranged accordingly.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm.

The bidiagonal SVD algorithm of an intermediate bidiagonal form B of MAT fails to converge if the number of QR sweeps exceeds MAXITER * min(m,n). Convergence usually occurs in about 2 * min(m,n) QR sweeps.

The default is 10.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm.

MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of min(m,n) and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm applied to the associated symmetric tridiagonal matrix B * B' whose eigenvalues are the squares of the singular values of B or MAT.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm applied to the associated symmetric tridiagonal matrix B * B' whose eigenvalues are the squares of the singular values of B of MAT.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

REORTHO (INPUT, OPTIONAL) logical(lgl) If the optional argument REORTHO is set to true, reorthogonalization is performed (when needed) in the Ralha-Barlow one-sided bidiagonalization of MAT. If this optional logical argument is set to false, reorthogonalization is never performed, which can result in a loss of orthogonality for the left (right) singular vectors of MAT if m>=n (m<n) and MAT has a large condition number.

See description of the BD_CMP2 subroutine for further details.

The default is REORTHO = true.

FAILURE_BD (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE_BD = false : indicates that maximum accuracy was obtained in the Ralha-Barlow one-sided bidiagonalization of MAT.
- FAILURE_BD = true : indicates that MAT is nearly singular and some loss of orthogonality can be expected in the Ralha-Barlow bidiagonalization algorithm.

See description of the BD_CMP2 subroutine for further details.

Further Details

Computing the SVD of a m-by-n matrix MAT with m>=n in subroutine SVD_CMP3 consists of three steps:

1) reduction of the rectangular matrix to bidiagonal form B via the Ralha-Barlow one-sided bidiagonalization algorithm, see the references (1) and (2);
2) in place accumulation of the orthogonal transformations used in the reduction to bidiagonal form B, see the references (3) and (4);
3) computation of the SVD of the bidiagonal matrix B, see the references (3) and (4).

In cases of a large condition number of MAT, this three-step algorithm may compute left singular vectors of MAT, which are not numerically orthogonal. This is because the left (orthogonal) matrix in the bidiagonal decomposition of MAT (estimated by the Ralha-Barlow one-sided bidiagonalization algorithm) may also not be numerically orthogonal as it is computed by a recurrence relationship, see the references (1) and (2) for details. To correct partly this deficiency, partial reorthogonalization can be performed to ensure orthogonality at the expense of speed of computation. The reorthogonalization uses the Gram-Schmidt method described in the reference (5) and is activated if the optional logical argument REORTHO is set to true.

However, the largest left singular vectors of MAT are always numerically orthogonal even if MAT is singular or nearly singular, see the reference (1).

If m<n, this three-step algorithm is applied to MAT', instead of MAT, to get the SVD of MAT and it computes also numerically orthogonal right singular vectors of MAT in that case.

Note that if max(m,n) is much larger than min(m,n), the rectangular matrix is first reduced to upper or lower triangular form by a QR or LQ factorization and the three-steps reduction algorithm is applied to the resulting triangular factor. The singular vectors of the rectangular matrix are then obtained from those of the triangular factor by a back-transformation algorithm.

For further details on the SVD of a rectangular matrix and the algorithms to compute it, see the references below.

The three or four steps of the SVD algorithm used here (e.g., preliminary QR or LQ factorization, reduction to bidiagonal form B, in place accumulation of the orthogonal transformations and computation of the SVD of the bidiagonal matrix B) are parallelized if OPENMP is used.

For more details, see:

(1) **Barlow, J.L., Bosner, N., and Drmac, Z., 2005:** A new stable bidiagonal reduction algorithm. Linear Algebra Appl., No 397, pp. 35-84.

(2) **Bosner, N., and Barlow, J.L., 2007:** Block and Parallel versions of one-sided bidiagonalization. SIAM J. Matrix Anal. Appl., Volume 29, No 3, pp. 927-953.

(3) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(4) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(5) **Stewart, G.W., 2007:** Block Gram-Schmidt Orthogonalization. Report TR-4823, Department of Computer Science, College Park, University of Maryland.

6.19.40 subroutine svd_cmp3 (mat, s, failure, sort, maxiter, bisect, dqds, save_mat, reortho, failure_bd)

Purpose

SVD_CMP3 computes the singular values of a real m-by-n matrix MAT. The singular value decomposition (SVD) is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative. The columns of U and V are, respectively, the left and right singular vectors of MAT.

The original matrix MAT is first reduced to upper or lower bidiagonal form B by an orthogonal transformation:

Q' * MAT * P = B

where Q and P are orthogonal. The singular values SIGMA of the bidiagonal matrix B are then computed by the bidiagonal implicit QR algorithm (if BISECT=false and DQDS=false), the bisection method (if BISECT=true) or the dqds algorithm (if DQDS=true) applied to the bidiagonal matrix B.

The routine returns only the first min(m,n) singular values of MAT.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, the m-by-n matrix MAT is destroyed if m>=n and the optional argument SAVE_MAT is not used with the value true.

S (OUTPUT) real(stnd), dimension(:) The singular values of MAT.

The size of S must verify: size(S) = min(m,n) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the implicit QR, bisection or dqds algorithm used to compute the singular values of the bidiagonal form B of the input m-by-n matrix MAT did not converge and that full accuracy was not attained in the bidiagonal SVD of this intermediate bidiagonal form B of MAT. The sign of the incorrect singular values is set to negative.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or into descending order if SORT = 'D' or 'd'.

The default is to sort the singular values into descending order.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal implicit QR phase of the SVD algorithm.

The bidiagonal SVD algorithm of an intermediate bidiagonal form B of MAT fails to converge if the number of QR sweeps exceeds MAXITER * min(m,n). Convergence usually occurs in about 2 * min(m,n) QR sweeps.

This argument has no effect if BISECT is equal to true.

The default is 10.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values of the intermediate min(m,n)-by-min(m,n) bidiagonal matrix B are computed.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slower execution time.

If BISECT is set to false, singular values are computed with the bidiagonal implicit QR or dqds algorithm applied to the associated min(m,n)-by-min(m,n) bidiagonal matrix B.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values of the intermediate min(m,n)-by-min(m,n) bidiagonal matrix B are computed.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition. Moreover, the dqds algorithm is usually also faster than the bisection or bidiagonal implicit QR algorithms and is the default method used in SVD_CMP3 for computing singular values of the bidiagonal form B of the input matrix MAT.

If DQDS is set to false, singular values are computed with the bidiagonal implicit QR or bisection algorithm applied to the associated min(m,n)-by-min(m,n) bidiagonal matrix B.

If both optional arguments BISECT and DQDS are specified with the value true, the dqds algorithm is used.

If both optional arguments BISECT and DQDS are specified with the value false, the bidiagonal implicit QR algorithm is used.

The default is true.

SAVE_MAT (INPUT, OPTIONAL) logical(lgl) On entry, if SAVE_MAT is set to true, the m-by-n matrix MAT is not modified by the routine.

The default is false.

REORTHO (INPUT, OPTIONAL) logical(lgl) If the optional argument REORTHO is set to true, reorthogonalization is performed (when needed) in the Ralha-Barlow one-sided bidiagonalization of MAT. If this optional logical argument is set to false, reorthogonalization is never performed, which can result in a loss of orthogonality for the left (right) singular vectors of MAT if m>=n (m<n) if MAT has a large condition number. However, this loss of orthogonality is not detrimental to the accuracy of the singular values.

See description of the BD_CMP2 subroutine for further details.

The default is REORTHO = false.

FAILURE_BD (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE_BD = false : indicates that maximum accuracy was obtained in the Ralha-Barlow one-sided bidiagonalization of MAT.
- FAILURE_BD = true : indicates that MAT is nearly singular.

Further Details

Computing the singular values of a rectangular matrix in subroutine SVD_CMP3 consists of two steps:

1) reduction of the rectangular matrix to bidiagonal form B via the Ralha-Barlow one-sided bidiagonalization algorithm, see the references (1) and (2);

2) computation of the singular values of the min(m,n)-by-min(m,n) bidiagonal matrix B by a bidiagonal implicit QR algorithm (if BISECT=false and DQDS=false), a dqds algorithm (if DQDS=true) or a bisection algorithm (if BISECT=true). See the references (3), (4), (5) and (6) for a description of these different algorithms.

Note that if max(m,n) is much larger than min(m,n), the rectangular matrix is first reduced to upper or lower triangular form by a QR or LQ factorization and the reduction algorithm is applied to the resulting triangular factor. The singular values of the rectangular matrix are then obtained from those of the triangular factor.

For further details on the SVD of a rectangular matrix and the algorithms to compute it, see the references below.

The different steps of the SVD algorithm used here (e.g., preliminary QR or LQ factorization, reduction to bidiagonal form, computation of the singular values of the bidiagonal form) are parallelized if OPENMP is used.

For more details, see:

(1) **Barlow, J.L., Bosner, N., and Drmac, Z., 2005:** A new stable bidiagonal reduction algorithm. Linear Algebra Appl., No 397, pp. 35-84.

(2) **Bosner, N., and Barlow, J.L., 2007:** Block and Parallel versions of one-sided bidiagonalization. SIAM J. Matrix Anal. Appl., Volume 29, No 3, pp. 927-953.

(3) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(4) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(5) **Fernando, K.V., and Parlett, B.N., 1994:** Accurate singular values and differential qd algorithms. Numer. Math., Volume 67, No 2, pp. 191-229.

(6) **Li, S., Gu, M., and Parlett, B. N., 2014:** An Improved DQDS Algorithm. SIAM Journal on Scientific Computing, Volume 36, No 3, C290-C308.

6.19.41 subroutine svd_cmp4 (mat, s, failure, v, sort, maxiter, max_francis_steps, perfect_shift, bisect, dqds, sing_vec, gen_p, reortho, failure_bd, d, e)

Purpose

SVD_CMP4 computes the Singular Value Decomposition (SVD) of a real m-by-n matrix MAT with m>=n. The SVD is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative. The columns of U and V are, respectively, the left and right singular vectors of MAT.

The routine returns the first n singular values and the associated left and right singular vectors.

MAT is first reduced to bidiagonal form B with the help of the Ralha-Barlow one-sided bidiagonalization algorithm, see the references (1) and (2).

The singular values and right singular vectors of B are then computed by the bidiagonal implicit QR algorithm applied to B, see the references (3) and (4).

The singular vectors of MAT are finally computed by a back transformation algorithm and an orthogonalization step for the left singular vectors.

Optionally, if the logical argument SING_VEC is used with the value false, matrices the routine computes only the singular values by the dqds algorithm and the orthogonal Q and P used to reduce MAT to bidiagonal form B. This is useful for computing a partial SVD of MAT with subroutines BD_INVITER2 or BD_DEFLATE2 for example.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit:

- if SING_VEC=true, MAT is overwritten with the first n columns of U (the left singular vectors, stored column-wise);
- if SING_VEC=false, MAT is overwritten with the first n columns of Q (stored column-wise), the orthogonal matrix used to reduce MAT to bidiagonal form as returned by subroutine BD_CMP2 in its argument MAT.

The shape of MAT must verify: size(MAT, 1) >= size(MAT, 2) = n .

S (OUTPUT) real(stnd), dimension(:) The singular values of MAT.

The size of S must verify: size(S) = size(MAT, 2) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the bidiagonal SVD or dqds algorithms did not converge and that full accuracy was not attained in the bidiagonal SVD of an intermediate bidiagonal form B of MAT.

V (OUTPUT) real(stnd), dimension(:,:) On exit:

- if SING_VEC=true, V is overwritten with the n-by-n orthogonal matrix V (the right singular vectors, stored column-wise);
- if SING_VEC=false, V is overwritten with the n-by-n orthogonal matrix P (stored column-wise or in packed form), the orthogonal matrix used to reduce MAT to bidiagonal form as returned by subroutine BD_CMP2 in its argument P.

The shape of V must verify: size(V, 1) = size(V, 2) = n .

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or into descending order if SORT = 'D' or 'd'. The singular vectors are rearranged accordingly.

The default is to sort the singular values and vectors into descending order.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm.

The bidiagonal SVD algorithm of an intermediate bidiagonal form B of MAT fails to converge if the number of QR sweeps exceeds MAXITER * min(m,n). Convergence usually occurs in about 2 * min(m,n) QR sweeps.

The default is 10.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) On entry, this optional argument has an effect only if the optional argument SING_VEC has the value true.

MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm.

MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of n and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) On entry, this optional argument has an effect only if the optional argument SING_VEC has the value true.

PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm applied to the associated symmetric tridiagonal matrix B * B' whose eigenvalues are the squares of the singular values of B or MAT.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm applied to the associated symmetric tridiagonal matrix B * B' whose eigenvalues are the squares of the singular values of B or MAT.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

SING_VEC (INPUT, OPTIONAL) logical(lgl) On entry:

- if SING_VEC=true, the routine computes the singular values and vectors of MAT.
- If SING_VEC=false the routine computes only the singular values of MAT and the orthogonal matrices Q and P used to reduce MAT to upper bidiagonal form as returned by subroutine BD_CMP2. See the description of BD_CMP2 subroutine for more details.

The default is true.

GEN_P (INPUT, OPTIONAL) logical(lgl) On entry, this optional argument has an effect only if the optional argument SING_VEC is also used with the value false.

In this case, if the optional argument GEN_P is used and is set to true, the orthogonal matrix P used to reduce MAT to bidiagonal form is generated on output of the subroutine in its argument V.

If this argument is set to false, the orthogonal matrix P is stored in factored form as products of elementary reflectors in the lower triangle of the array V.

See the description of BD_CMP2 subroutine for more details.

The default is true.

REORTHO (INPUT, OPTIONAL) logical(lgl) On entry, this optional argument has an effect only if the optional argument SING_VEC is also used with the value false.

In this case, if the optional argument REORTHO is set to true, reorthogonalization is performed (when needed) in the Ralha-Barlow one-sided bidiagonalization of MAT. If this optional logical argument is set to false, reorthogonalization is never performed, which can result in a loss of orthogonality if MAT has a large condition number.

See description of the BD_CMP2 subroutine for further details.

The default is REORTHO = true.

FAILURE_BD (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE_BD = false : indicates that maximum accuracy was obtained in the Ralha-Barlow one-sided bidiagonalization of MAT;
- FAILURE_BD = true : indicates that MAT is nearly singular and some loss of orthogonality can be expected in the left orthogonal matrix in the bidiagonal reduction of MAT obtained by the one-sided Rhala-Barlow algorithm if SING_VEC=false is used in the call to SVD_CMP4.

See description of the BD_CMP2 subroutine for further details.

D (OUTPUT, OPTIONAL) real(stnd), dimension(:) The diagonal elements of the intermediate upper bidiagonal matrix B.

The size of D must be size(D) = size(MAT, 2) = n .

E (OUTPUT, OPTIONAL) real(stnd), dimension(:) The off-diagonal elements of the intermediate upper bidiagonal matrix B:

E(i) = B(i-1,i) for i = 2,3,. . . ,n;

E(1) is arbitrary.

The size of E must be size(E) = size(MAT, 2) = n .

Further Details

Computing the SVD of a m-by-n matrix MAT, with m>=n, in subroutine SVD_CMP4 consists of four steps:

1) reduction of the rectangular matrix to bidiagonal form B via the Ralha-Barlow one-sided bidiagonalization algorithm, see the references (1) and (2);

2) in place accumulation of the right orthogonal transformations used in the reduction of MAT to bidiagonal form B, see the references (3) and (4);

3) computation of the singular values and right singular vectors of MAT by applying the implicit QR algorithm to the bidiagonal matrix B, see the references (3) and (4);

4) computation and orthogonalization of the left singular vectors in the SVD of MAT to avoid the possible loss of orthogonality of the left orthogonal matrix in the bidiagonal factorization of MAT computed by the one-sided bidiagonalization algorithm, see the references (3) and (4).

This four-step algorithm computes numerically orthogonal left singular vectors of MAT even in cases of large condition number of MAT despite that the left (orthogonal) matrix in the bidiagonal decomposition of MAT computed by the Ralha-Barlow one-sided bidiagonalization algorithm may not be numerically orthogonal if MAT is nearly singular or has a very large condition number (see references (1) and (2) for details).

If singular vectors are requested (e.g., if the optional logical argument SING_VEC is not used or is used and set to true) and max(m,n) is much larger than min(m,n), the rectangular matrix is first reduced to upper triangular form by a QR factorization and the above four-step algorithm is applied to the resulting triangular factor. The left singular vectors of the rectangular matrix are then obtained from those of the triangular factor by a back-transformation algorithm.

For further details on the SVD of a rectangular matrix and the different algorithms to compute it, see the references below.

The four or five steps of the SVD algorithm used here (e.g., preliminary QR or LQ factorization, the reduction to bidiagonal form B, in place accumulation of the orthogonal transformations, computation of the SVD of the bidiagonal matrix B and computation/orthogonalization of the left singular vectors in the SVD of MAT) are parallelized if OPENMP is used.

Optionally, the intermediate bidiagonal decomposition of MAT can be output by the subroutine if the optional logical argument SING_VEC is used with the value false and the optional arguments D and E are also specified. Note, however, that in that case the left (orthogonal) matrix in the bidiagonal decomposition of MAT (computed by the Ralha-Barlow one-sided bidiagonalization algorithm) may not be numerically orthogonal if MAT is nearly singular or has a very large condition number (see references (1) and (2) for details). To correct partly this deficiency, partial reorthogonalization can be performed to ensure orthogonality of this left matrix at the expense of speed of computation. The reorthogonalization uses the Gram-Schmidt method described in the reference (5) and is activated if the optional logical argument REORTHO is set to true.

When the optional logical argument SING_VEC is used with the value false, the singular values of the bidiagonal form B of MAT are computed by the more accurate and faster dqds algorithm. See references (6) and (7) for a description of the dqds algorithm.

For more details, see:

(1) **Barlow, J.L., Bosner, N., and Drmac, Z., 2005:** A new stable bidiagonal reduction algorithm. Linear Algebra Appl., No 397, pp. 35-84.

(2) **Bosner, N., and Barlow, J.L., 2007:** Block and Parallel versions of one-sided bidiagonalization. SIAM J. Matrix Anal. Appl., Volume 29, No 3, pp. 927-953.

(3) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(4) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(5) **Stewart, G.W., 2007:** Block Gram-Schmidt Orthogonalization. Report TR-4823, Department of Computer Science, College Park, University of Maryland.

(6) **Fernando, K.V., and Parlett, B.N., 1994:** Accurate singular values and differential qd algorithms. Numer. Math., Volume 67, No 2, pp. 191-229.

(7) **Li, S., Gu, M., and Parlett, B. N., 2014:** An Improved DQDS Algorithm. SIAM Journal on Scientific Computing, Volume 36, No 3, C290-C308.

6.19.42 subroutine svd_cmp4 (mat, s, failure, sort, maxiter, bisect, dqds, save_mat, reortho, failure_bd)

Purpose

SVD_CMP4 computes the singular values of a real m-by-n matrix MAT. The singular value decomposition (SVD) is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative. The columns of U and V are, respectively, the left and right singular vectors of MAT.

The original matrix MAT is first reduced to upper or lower bidiagonal form B by an orthogonal transformation:

Q' * MAT * P = B

where Q and P are orthogonal. The singular values SIGMA of the bidiagonal matrix B are then computed by the bidiagonal implicit QR algorithm (if BISECT=false and DQDS=false), the bisection method (if BISECT=true) or the dqds algorithm (if DQDS=true) applied to the bidiagonal matrix B.

The routine returns only the first min(m,n) singular values of MAT.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, the m-by-n matrix MAT is destroyed if m>=n and the optional argument SAVE_MAT is not used with the value true.

S (OUTPUT) real(stnd), dimension(:) The singular values of MAT.

The size of S must verify: size(S) = min(m,n) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the implicit QR, bisection or dqds algorithm used to compute the singular values of the bidiagonal form B of the input m-by-n matrix MAT did not converge and that full accuracy was not attained in the bidiagonal SVD of this intermediate bidiagonal form B of MAT. The sign of the incorrect singular values is set to negative.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or into descending order if SORT = 'D' or 'd'.

The default is to sort the singular values into descending order.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal implicit QR phase of the SVD algorithm.

The bidiagonal SVD algorithm of an intermediate bidiagonal form B of MAT fails to converge if the number of QR sweeps exceeds MAXITER * min(m,n). Convergence usually occurs in about 2 * min(m,n) QR sweeps.

This argument has no effect if BISECT is equal to true.

The default is 10.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values of the intermediate min(m,n)-by-min(m,n) bidiagonal matrix B are computed.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slower execution time.

If BISECT is set to false, singular values are computed with the bidiagonal implicit QR or dqds algorithm applied to the associated min(m,n)-by-min(m,n) bidiagonal matrix B.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values of the intermediate min(m,n)-by-min(m,n) bidiagonal matrix B are computed.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition. Moreover, the dqds algorithm is usually also faster than the bisection or bidiagonal implicit QR algorithms and is the default method used in SVD_CMP4 for computing singular values of the bidiagonal form B of the input matrix MAT.

If DQDS is set to false, singular values are computed with the bidiagonal implicit QR or bisection algorithm applied to the associated min(m,n)-by-min(m,n) bidiagonal matrix B.

If both optional arguments BISECT and DQDS are specified with the value true, the dqds algorithm is used.

If both optional arguments BISECT and DQDS are specified with the value false, the bidiagonal implicit QR algorithm is used.

The default is true.

SAVE_MAT (INPUT, OPTIONAL) logical(lgl) On entry, if SAVE_MAT is set to true, the m-by-n matrix MAT is not modified by the routine.

The default is false.

REORTHO (INPUT, OPTIONAL) logical(lgl) If the optional argument REORTHO is set to true, reorthogonalization is performed (when needed) in the Ralha-Barlow one-sided bidiagonalization of MAT. If this optional logical argument is set to false, reorthogonalization is never performed, which can result in a loss of orthogonality for the left (right) singular vectors of MAT if m>=n (m<n) if MAT has a large condition number. However, this loss of orthogonality is not detrimental to the accuracy of the singular values.

See description of the BD_CMP2 subroutine for further details.

The default is REORTHO = false.

FAILURE_BD (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE_BD = false : indicates that maximum accuracy was obtained in the Ralha-Barlow one-sided bidiagonalization of MAT.
- FAILURE_BD = true : indicates that MAT is nearly singular.

Further Details

Computing the singular values of a rectangular matrix in subroutine SVD_CMP4 consists of two steps:

1) reduction of the rectangular matrix to bidiagonal form B via the Ralha-Barlow one-sided bidiagonalization algorithm, see the references (1) and (2);

2) computation of the singular values of the min(m,n)-by-min(m,n) bidiagonal matrix B by a bidiagonal implicit QR algorithm (if BISECT=false and DQDS=false), a dqds algorithm (if DQDS=true) or a

bisection algorithm (if BISECT=true). See the references (3), (4), (5) and (6) for a description of these different algorithms.

Note that if max(m,n) is much larger than min(m,n), the rectangular matrix is first reduced to upper or lower triangular form by a QR or LQ factorization and the reduction algorithm is applied to the resulting triangular factor. The singular values of the rectangular matrix are then obtained from those of the triangular factor.

For further details on the SVD of a rectangular matrix and the algorithms to compute it, see the references below.

The different steps of the SVD algorithm used here (e.g., preliminary QR or LQ factorization, reduction to bidiagonal form, computation of the singular values of the bidiagonal form) are parallelized if OPENMP is used.

For more details, see:

(1) **Barlow, J.L., Bosner, N., and Drmac, Z., 2005:** A new stable bidiagonal reduction algorithm. Linear Algebra Appl., No 397, pp. 35-84.

(2) **Bosner, N., and Barlow, J.L., 2007:** Block and Parallel versions of one-sided bidiagonalization. SIAM J. Matrix Anal. Appl., Volume 29, No 3, pp. 927-953.

(3) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(4) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(5) **Fernando, K.V., and Parlett, B.N., 1994:** Accurate singular values and differential qd algorithms. Numer. Math., Volume 67, No 2, pp. 191-229.

(6) **Li, S., Gu, M., and Parlett, B. N., 2014:** An Improved DQDS Algorithm. SIAM Journal on Scientific Computing, Volume 36, No 3, C290-C308.

6.19.43 subroutine svd_cmp5 (mat, s, failure, v, sort, maxiter, max_francis_steps, perfect_shift, bisect, dqds, use_qr, failure_bd)

Purpose

SVD_CMP5 computes the Singular Value Decomposition (SVD) of a real m-by-n matrix MAT. The SVD is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative. The columns of U and V are, respectively, the left and right singular vectors of MAT.

The routine returns the first min(m,n) singular values and the associated left and right singular vectors.

MAT (or MAT' if m<n) is first reduced to bidiagonal form B with the help of the Ralha-Barlow one-sided bidiagonalization algorithm, see the references (1) and (2).

The singular values and right (left if m<n) singular vectors of B are then computed by the bidiagonal implicit QR algorithm applied to B, see the references (3) and (4).

The singular vectors of MAT are finally computed by a back transformation algorithm and and an orthogonalization step for the left (right if m<n) singular vectors.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is overwritten with the first min(m,n) columns of U, the left singular vectors.

S (OUTPUT) real(stnd), dimension(:) The singular values of MAT.

The size of S must verify: size(S) = min(m,n) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the bidiagonal SVD algorithm did not converge and that full accuracy was not attained in the bidiagonal SVD of an intermediate bidiagonal form B of MAT.

V (OUTPUT) real(stnd), dimension(:,:) On exit, V contains the first min(m,n) columns of V, the right singular vectors.

The shape of V must verify:

- size(V, 1) = n,
- size(V, 2) = min(m,n).

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or into descending order if SORT = 'D' or 'd'. The singular vectors are rearranged accordingly.

The default is to sort the singular values and vectors into descending order.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm.

The bidiagonal SVD algorithm of an intermediate bidiagonal form B of MAT fails to converge if the number of QR sweeps exceeds MAXITER * min(m,n). Convergence usually occurs in about 2 * min(m,n) QR sweeps.

The default is 10.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm.

MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of min(m,n) and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm applied to the associated symmetric tridiagonal matrix B * B' whose eigenvalues are the squares of the singular values of B or MAT.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm applied to the associated symmetric tridiagonal matrix B * B' whose eigenvalues are the squares of the singular values of B or MAT.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

USE_QR (INPUT, OPTIONAL) logical(lgl) USE_QR determines if a preliminary QR or LQ factorization is used first to reduce the matrix MAT to triangular form before computing its SVD.

If USE_QR is set to true and max(m,n) exceeds THR_BD3*min(m,n), where THR_BD3 is a module parameter defined in module SVD_Procedures, a QR or LQ factorization is used before computing the SVD of MAT.

The default is true.

FAILURE_BD (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE_BD = false : indicates that maximum accuracy was obtained in the Ralha-Barlow one-sided bidiagonalization of MAT.
- FAILURE_BD = true : indicates that MAT is nearly singular.

Further Details

Computing the SVD of a m-by-n matrix MAT, with m>=n, in subroutine SVD_CMP5 consists of four steps:

1) reduction of the rectangular matrix to bidiagonal form B via the Ralha-Barlow one-sided bidiagonalization algorithm, see the references (1) and (2);

2) in place accumulation of the right orthogonal transformations used in the reduction of MAT to bidiagonal form B, see the references (3) and (4);

3) computation of the singular values and right singular vectors of MAT by applying the implicit QR algorithm to the bidiagonal matrix B, see the references (3) and (4);

4) computation and orthogonalization of the left singular vectors in the SVD of MAT to avoid the possible loss of orthogonality of the left orthogonal matrix in the bidiagonal factorization of MAT computed by the one-sided bidiagonalization algorithm, see the references (1) and (2).

This four-step algorithm computes numerically orthogonal left singular vectors of a m-by-n matrix MAT, with m>=n, even in cases of large condition number of MAT despite that the left (orthogonal) matrix in the

bidiagonal decomposition of MAT computed by the Ralha-Barlow one-sided bidiagonalization algorithm may not be numerically orthogonal if MAT is nearly singular or has a very large condition number (see references (1) and (2) for details).

If m<n, this four-step algorithm is applied to MAT', instead of MAT, to get the SVD of MAT and it computes also numerically orthogonal right singular vectors of MAT in that case.

Note that, if max(m,n) is much larger than min(m,n), at the user option, the rectangular matrix can be first reduced to upper or lower triangular form by a QR or LQ factorization and the four-step reduction algorithm is applied to the resulting triangular factor. The singular vectors of the rectangular matrix are then obtained from those of the triangular factor in the QR or LQ factorization by a back-transformation algorithm (see the references (3) and (4) for details).

For further details on the SVD of a rectangular matrix and the algorithms to compute it, see the references below.

The four or five steps of the SVD algorithm used here (e.g., preliminary QR or LQ factorization, reduction to bidiagonal form B, in place accumulation of the orthogonal transformations, computation of the SVD of the bidiagonal matrix B and reorthogonalization of the left (or right if m<n) singular vectors in the SVD of MAT) are parallelized if OPENMP is used.

For more details, see:

(1) **Barlow, J.L., Bosner, N., and Drmac, Z., 2005:** A new stable bidiagonal reduction algorithm. Linear Algebra Appl., No 397, pp. 35-84.

(2) **Bosner, N., and Barlow, J.L., 2007:** Block and Parallel versions of one-sided bidiagonalization. SIAM J. Matrix Anal. Appl., Volume 29, No 3, pp. 927-953.

(3) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(4) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

6.19.44 subroutine svd_cmp5 (mat, s, failure, sort, maxiter, bisect, dqds, save_mat, reortho, failure_bd)

Purpose

SVD_CMP5 computes the singular values of a real m-by-n matrix MAT. The singular value decomposition (SVD) is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative. The columns of U and V are, respectively, the left and right singular vectors of MAT.

The original matrix MAT is first reduced to upper or lower bidiagonal form B by an orthogonal transformation:

Q' * MAT * P = B

where Q and P are orthogonal. The singular values SIGMA of the bidiagonal matrix B are then computed by the bidiagonal implicit QR algorithm (if BISECT=false and DQDS=false), the bisection method (if BISECT=true) or the dqds algorithm (if DQDS=true) applied to the bidiagonal matrix B.

The routine returns only the first min(m,n) singular values of MAT.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, the m-by-n matrix MAT is destroyed if m>=n and the optional argument SAVE_MAT is not used with the value true.

S (OUTPUT) real(stnd), dimension(:) The singular values of MAT.

The size of S must verify: size(S) = min(m,n) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the implicit QR, bisection or dqds algorithm used to compute the singular values of the bidiagonal form B of the input m-by-n matrix MAT did not converge and that full accuracy was not attained in the bidiagonal SVD of this intermediate bidiagonal form B of MAT. The sign of the incorrect singular values is set to negative.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or into descending order if SORT = 'D' or 'd'.

The default is to sort the singular values into descending order.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal implicit QR phase of the SVD algorithm.

The bidiagonal SVD algorithm of an intermediate bidiagonal form B of MAT fails to converge if the number of QR sweeps exceeds MAXITER * min(m,n). Convergence usually occurs in about 2 * min(m,n) QR sweeps.

This argument has no effect if BISECT is equal to true.

The default is 10.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values of the intermediate min(m,n)-by-min(m,n) bidiagonal matrix B are computed.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slower execution time.

If BISECT is set to false, singular values are computed with the bidiagonal implicit QR or dqds algorithm applied to the associated min(m,n)-by-min(m,n) bidiagonal matrix B.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values of the intermediate min(m,n)-by-min(m,n) bidiagonal matrix B are computed.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition. Moreover, the dqds algorithm is usually also faster than the bisection or bidiagonal implicit QR algorithms and is the default method used in SVD_CMP5 for computing singular values of the bidiagonal form B of the input matrix MAT.

If DQDS is set to false, singular values are computed with the bidiagonal implicit QR or bisection algorithm applied to the associated min(m,n)-by-min(m,n) bidiagonal matrix B.

If both optional arguments BISECT and DQDS are specified with the value true, the dqds algorithm is used.

If both optional arguments BISECT and DQDS are specified with the value false, the bidiagonal implicit QR algorithm is used.

The default is true.

SAVE_MAT (INPUT, OPTIONAL) logical(lgl) On entry, if SAVE_MAT is set to true, the m-by-n matrix MAT is not modified by the routine.

The default is false.

REORTHO (INPUT, OPTIONAL) logical(lgl) If the optional argument REORTHO is set to true, reorthogonalization is performed (when needed) in the Ralha-Barlow one-sided bidiagonalization of MAT. If this optional logical argument is set to false, reorthogonalization is never performed, which can result in a loss of orthogonality for the left (right) singular vectors of MAT if m>=n (m<n) if MAT has a large condition number. However, this loss of orthogonality is not detrimental to the accuracy of the singular values.

See description of the BD_CMP2 subroutine for further details.

The default is REORTHO = false.

FAILURE_BD (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE_BD = false : indicates that maximum accuracy was obtained in the Ralha-Barlow one-sided bidiagonalization of MAT.
- FAILURE_BD = true : indicates that MAT is nearly singular.

Further Details

Computing the singular values of a rectangular matrix in subroutine SVD_CMP5 consists of two steps:

1) reduction of the rectangular matrix to bidiagonal form B via the Ralha-Barlow one-sided bidiagonalization algorithm, see the references (1) and (2);
2) computation of the singular values of the min(m,n)-by-min(m,n) bidiagonal matrix B by a bidiagonal implicit QR algorithm (if BISECT=false and DQDS=false), a dqds algorithm (if DQDS=true) or a bisection algorithm (if BISECT=true). See the references (3), (4), (5) and (6) for a description of these different algorithms.

Note that if max(m,n) is much larger than min(m,n), the rectangular matrix is first reduced to upper or lower triangular form by a QR or LQ factorization and the reduction algorithm is applied to the resulting triangular factor. The singular values of the rectangular matrix are then obtained from those of the triangular factor.

For further details on the SVD of a rectangular matrix and the algorithms to compute it, see the references below.

The different steps of the SVD algorithm used here (e.g., preliminary QR or LQ factorization, reduction to bidiagonal form, computation of the singular values of the bidiagonal form) are parallelized if OPENMP is used.

For more details, see:

(1) **Barlow, J.L., Bosner, N., and Drmac, Z., 2005:** A new stable bidiagonal reduction algorithm. Linear Algebra Appl., No 397, pp. 35-84.

(2) **Bosner, N., and Barlow, J.L., 2007:** Block and Parallel versions of one-sided bidiagonalization. SIAM J. Matrix Anal. Appl., Volume 29, No 3, pp. 927-953.

(3) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(4) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(5) **Fernando, K.V., and Parlett, B.N., 1994:** Accurate singular values and differential qd algorithms. Numer. Math., Volume 67, No 2, pp. 191-229.

(6) **Li, S., Gu, M., and Parlett, B. N., 2014:** An Improved DQDS Algorithm. SIAM Journal on Scientific Computing, Volume 36, No 3, C290-C308.

6.19.45 subroutine svd_cmp6 (mat, s, v, failure, sort, nsvd, maxiter, ortho, backward_sweep, scaling, initvec, use_qr, failure_bd, failure_bisect)

Purpose

SVD_CMP6 computes a full or partial Singular Value Decomposition (SVD) of a real m-by-n matrix MAT. The full SVD is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative. The columns of U and V are, respectively, the left and right singular vectors of MAT.

The routine can return the first min(m,n) singular values and the associated left and right singular vectors or a truncated SVD if the optional integer parameter NSVD is used in the call to SVD_CMP6.

MAT (or MAT' if m<n) is first reduced to bidiagonal form B with the help of the Ralha-Barlow one-sided bidiagonalization algorithm without reorthogonalisation, see the references (1) and (2).

The singular values and right (left if m<n) singular vectors of B are then computed by the bisection and inverse iteration methods applied to B and the tridiagonal matrix B' * B, respectively, see the reference (3).

The singular vectors of MAT are finally computed by a back transformation algorithm and an orthogonalization step for the left (right if m<n) singular vectors.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is overwritten with the first nsvd columns of U, e.g., the left singular vectors associated with the first nsvd largest singular values of MAT.

S (OUTPUT) real(stnd), dimension(:), pointer On exit, S(:) contains estimates of the first nsvd largest singular values of MAT. The singular values are given in decreasing order and are positive or zero.

The statut of the pointer S must not be undefined on entry. If, on entry, the pointer S is already allocated, it will be first deallocated and then reallocated with the correct size.

On exit, the size of the pointer S will verify:

- size(S) = nsvd <= min(size(MAT,1) , size(MAT,2)) = min(m,n) .

V (OUTPUT) real(stnd), dimension(:,:), pointer On exit, the computed first nsvd columns of V, e.g., the right singular vectors associated with the first nsvd largest singular values of MAT.

The right singular vector associated with the singular value S(j) is stored in the j-th column of V.

The statut of the pointer V must not be undefined on entry. If, on entry, the pointer V is already allocated, it will be first deallocated and then reallocated with the correct shape.

On exit, the shape of the pointer V will verify:

- size(V, 1) = size(MAT, 2) = n ,
- size(V, 2) = size(S) = nsvd .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false indicates successful exit.
- FAILURE = true indicates that some singular vectors failed to converge in MAXITER inverse iterations.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or into descending order if SORT = 'D' or 'd'. The singular vectors are rearranged accordingly.

The default is to sort the singular values and vectors into descending order.

NSVD (INPUT/OUTPUT, OPTIONAL) integer(i4b) On entry, NSVD specifies the number of the top singular triplets which are requested.

On exit, NSVD is the number of singular triplets which have been computed by the subroutine, which can be greater than the requested number if multiple singular values at index NSVD make unique selection impossible.

On entry, NSVD must be greater than 0 and less or equal to min(m,n).

The default is NSVD = min(size(MAT,1) , size(MAT,2)) = min(m,n).

MAXITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed for computing singular vectors.

By default, 2 inverse iterations are performed for all the singular vectors.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, all the singular vectors are orthogonalized by the Modified Gram-Schmidt or QR algorithm in the inverse iteration algorithm;
- ORTHO=false, the singular vectors are not orthogonalized by the Modified Gram-Schmidt or QR algorithm in the inverse iteration algorithm.

The default is to orthogonalize the singular vectors only for the singular values, which are not well-separated.

BACKWARD_SWEEP (INPUT, OPTIONAL) logical(lgl) On entry, if:

- BACKWARD_SWEEP=true and the singular vectors are orthogonalized by the modified Gram-Schmidt algorithm in the inverse iteration algorithm, a backward sweep of the modified Gram-Schmidt algorithm is also performed;
- BACKWARD_SWEEP=false a backward sweep is not performed.

The default is not to perform a backward sweep of the modified Gram-Schmidt algorithm.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if:

- SCALING=true, the bidiagonal matrix B and tridiagonal matrix B' * B are scaled before computing the singular values and vectors, respectively;
- SCALING=false, the bidiagonal matrix B and tridiagonal matrix B' * B are not scaled.

The default is to scale the bidiagonal and tridiagonal matrices.

INITVEC (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITVEC=true, Fernando vectors are used to start the inverse iteration process for computing the singular vectors;
- INITVEC=false, random uniform starting vectors are used.

For unreduced tridiagonal matrices B' * B, the default is to use Fernando starting vectors if the eigenvalues (e.g., the squares of the singular values) are well-separated and random uniform starting vectors otherwise.

For reduced tridiagonal matrices B' * B, the default is to use random uniform starting vectors.

USE_QR (INPUT, OPTIONAL) logical(lgl) USE_QR determines if a preliminary QR or LQ factorization is used first to reduce the matrix MAT to triangular form before computing its thin or truncated SVD.

If USE_QR is set to true and max(m,n) exceeds THR_BD3*min(m,n), where THR_BD3 is a module parameter defined in module SVD_Procedures, a QR or LQ factorization is used before computing the SVD of MAT.

The default is true.

FAILURE_BD (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE_BD = false : indicates that maximum accuracy was obtained in the Ralha-Barlow one-sided bidiagonalization of MAT;
- FAILURE_BD = true : indicates that MAT is nearly singular.

FAILURE_BISECT (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE_BISECT = false : indicates successful exit and the bisection algorithm converged for all the computed singular values to the desired accuracy;
- FAILURE_BISECT = true : indicates that some or all of the singular values failed to converge or were not computed. This is generally caused by unexpectedly inaccurate arithmetic.

Further Details

Computing the SVD of a m-by-n matrix MAT, with m>=n, in subroutine SVD_CMP6 consists of four steps:

1) reduction of the rectangular matrix to bidiagonal form B via the Ralha-Barlow one-sided bidiagonalization algorithm, see the references (1) and (2);
2) computation of the singular values and right singular vectors of B by applying the bisection and inverse iteration algorithms to the matrices B and B' * B, respectively, see the reference (3);
3) computation of the right singular vectors of MAT from those of B by a back-transformation algorithm, see the references (3) and (4);
4) computation and orthogonalization of the left singular vectors in the SVD of MAT to avoid the possible loss of orthogonality of the left orthogonal matrix in the bidiagonal factorization of MAT computed by the one-sided bidiagonalization algorithm, see the references (1) and (2).

This four-step algorithm computes numerically orthogonal left singular vectors of a m-by-n matrix MAT, with m>=n, even in cases of large condition number of MAT despite that the left (orthogonal) matrix in the bidiagonal decomposition of MAT (computed by the Ralha-Barlow one-sided bidiagonalization algorithm) may not be numerically orthogonal when MAT is nearly singular or has a very large condition number (see references (1) and (2) for details).

If m<n, this four-step algorithm is applied to MAT', instead of MAT, to get the SVD of MAT and it computes also numerically orthogonal right singular vectors of MAT in that case.

Note that, if max(m,n) is much larger than min(m,n), at the user option, the rectangular matrix can be first reduced to upper or lower triangular form by a QR or LQ factorization and the four-step reduction algorithm is applied to the resulting triangular factor. The singular vectors of the rectangular matrix are then obtained from those of the triangular factor in the QR or LQ factorization by a back-transformation algorithm (see the references (3) and (4) for details).

For further details on the SVD of a rectangular matrix and the algorithms to compute it, see the references below.

The four or five steps of the SVD algorithm used here (e.g., preliminary QR or LQ factorization, reduction to bidiagonal form B, computation of the singular values and right (or left if m<n) singular vectors of B, computation of the right (or left if m<n) singular vectors of MAT and orthogonalization of the left (or right if m<n) singular vectors in the SVD of MAT) are parallelized if OPENMP is used.

For more details, see:

(1) **Barlow, J.L., Bosner, N., and Drmac, Z., 2005:** A new stable bidiagonal reduction algorithm. Linear Algebra Appl., No 397, pp. 35-84.

(2) **Bosner, N., and Barlow, J.L., 2007:** Block and Parallel versions of one-sided bidiagonalization. SIAM J. Matrix Anal. Appl., Volume 29, No 3, pp. 927-953.

(3) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(4) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

6.19.46 subroutine svd_cmp7 (mat, s, u, v, failure, sort, maxiter, ortho, backward_sweep, scaling, initvec, failure_dqds)

Purpose

SVD_CMP7 computes a full or partial Singular Value Decomposition (SVD) of a real m-by-n matrix MAT. The full SVD is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative. The columns of U and V are, respectively, the left and right singular vectors of MAT.

The routine returns the first min(m,n) singular values and the associated left and right singular vectors (e.g. the left and right singular vectors of MAT in the thin SVD of MAT) or a truncated partial SVD at the user option.

MAT is first reduced to bidiagonal form B with a one-stage or two-stage algorithm, depending on its shape, see Further Details below and the references (1) and (2).

The singular values and vectors of B are then computed by the dqds and inverse iteration methods applied to B, see the references (1), (3) and (4).

The singular vectors of MAT are finally computed by a blocked back-transformation algorithm.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is overwritten as it is used as workspace inside the subroutine.

S (OUTPUT) real(stnd), dimension(:) The singular values of MAT.

The size of S must verify: size(S) = min(m,n) .

U (OUTPUT) real(stnd), dimension(:,:) On exit, the computed first nsvd columns of U, e.g., the left singular vectors associated with the first nsvd largest singular values of MAT.

The left singular vector associated with the singular value S(j) is stored in the j-th column of U.

The shape of V must verify:

- size(U, 1) = size(MAT, 1) = m ,
- size(U, 2) = nsvd <= size(S) = min(m,n) .

V (OUTPUT) real(stnd), dimension(:,:) On exit, the computed first nsvd columns of V, e.g., the right singular vectors associated with the first nsvd largest singular values of MAT.

The right singular vector associated with the singular value S(j) is stored in the j-th column of V.

The shape of V must verify:

- size(V, 1) = size(MAT, 2) = n ,
- size(V, 2) = nsvd <= size(S) = min(m,n) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false indicates successful exit.
- FAILURE = true indicates that the dqds algorithm failed to converge for some singular values or that some singular vectors failed to converge in MAXITER inverse iterations. If the dqds algorithm failed to compute some singular values, the sign of the incorrect singular values in S is set to negative and arguments U and V are filled with a quiet NAN in output of the subroutine.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or into descending order if SORT = 'D' or 'd'. The singular vectors are rearranged accordingly.

The default is to sort the singular values and vectors into descending order.

MAXITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed for computing singular vectors.

By default, 2 inverse iterations are performed for all the singular vectors.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, all the singular vectors are orthogonalized by the Modified Gram-Schmidt or QR algorithm in the inverse iteration algorithm;
- ORTHO=false, the singular vectors are not orthogonalized by the Modified Gram-Schmidt or QR algorithm in the inverse iteration algorithm.

The default is to orthogonalize the singular vectors only for the singular values, which are not well-separated.

BACKWARD_SWEEP (INPUT, OPTIONAL) logical(lgl) On entry, if:

- BACKWARD_SWEEP=true and the singular vectors are orthogonalized by the modified Gram-Schmidt algorithm in the inverse iteration algorithm, a backward sweep of the modified Gram-Schmidt algorithm is also performed;

- BACKWARD_SWEEP=false, a backward sweep is not performed.

The default is not to perform a backward sweep of the modified Gram-Schmidt algorithm.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if:

- SCALING=true, the bidiagonal matrix B is scaled before computing the singular values and vectors;
- SCALING=false, the bidiagonal matrix B is not scaled.

The default is to scale the bidiagonal matrix.

INITVEC (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITVEC=true, Fernando vectors are used to start the inverse iteration process for computing the singular vectors of the bidiagonal matrix BD (e.g. the eigenvectors of the associated Golub-Kahan tridiagonal matrix);
- INITVEC=false, random uniform starting vectors are used.

The default is to use Fernando starting vectors if the singular values are well-separated and the Golub-Kahan form of the input bidiagonal matrix is unreduced, and random uniform starting vectors otherwise.

FAILURE_DQDS (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE_DQDS = false : indicates successful exit and the dqds algorithm converged for all the computed singular values to the desired accuracy;
- FAILURE_DQDS = true : indicates that some or all of the singular values failed to converge or were not computed. This is generally caused by unexpectedly inaccurate arithmetic. In that case, U and V are filled with a quiet NAN and the sign of the incorrect singular values in S is set to negative in output of the subroutine.

Further Details

Computing the SVD of a m-by-n matrix MAT in subroutine SVD_CMP7 consists of three steps:

1) reduction of the rectangular matrix to bidiagonal form B via orthogonal transformations (e.g. Householder transformations);
2) computation of the SVD of the bidiagonal matrix B with the dqds algorithm for the singular values and inverse iteration for all or the leading singular vectors at the user option;
3) computation of the singular vectors of MAT from those of its bidiagonal form B by a blocked back-transformation algorithm, see the references (1) and (2).

Note that if max(m,n) is much larger than min(m,n), the rectangular matrix is first reduced to upper or lower triangular form by a preliminary QR or LQ factorization and the three-step reduction algorithm described above is applied to the resulting triangular factor. The singular vectors of the original rectangular matrix are then obtained from those of the triangular factor in the QR or LQ factorization by another blocked back-transformation algorithm.

Concerning the inverse iteration algorithm used here, note that a first estimate of the singular vectors is computed by the Fernando method applied to the tridiagonal Golub-Kahan matrix associated with the bidiagonal matrix B (see the references (5) and (6) for details) for the singular values which are well-separated and if the Golub-Kahan form of the input bidiagonal matrix is unreduced. For the other singular values, a random start is used as a first estimate of the singular vectors as in the standard inverse-iteration algorithm.

The singular vectors of B are then computed or refined using inverse iteration on the tridiagonal Golub-Kahan matrix for all the singular values at one step.

By default, the singular vectors of B are then orthogonalized by the Modified Gram-Schmidt or QR algorithm only if the singular values are not well-separated.

Excepted for the computation of the singular values by the dqds algorithm, all the steps of the SVD algorithm used here (e.g., preliminary QR or LQ factorization, reduction to bidiagonal form B, computation of the singular vectors vectors of B, computation of the singular vectors of MAT by blocked back-transformation algorithms are parallelized if OPENMP is used.

For more details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Parlett, B.N., and Marques, O.A., 2000:** An implementation of the dqds algorithm (positive case). Linear Algebra Appl., Volume 309, pp. 217-259.

(4) **Li, S., Gu, M., and Parlett, B. N., 2014:** An Improved DQDS Algorithm. SIAM Journal on Scientific Computing, Volume 36, No 3, C290-C308.

(5) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(6) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

6.19.47 subroutine svd_cmp8 (mat, s, u, v, failure, sort, maxiter, ortho, backward_sweep, scaling, initvec, reortho, failure_bd, failure_dqds)

Purpose

SVD_CMP8 computes a full or partial Singular Value Decomposition (SVD) of a real m-by-n matrix MAT with m>=n. The full SVD is written:

MAT = U * SIGMA * V'

where SIGMA is an m-by-n matrix which is zero except for its n-diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of MAT; they are real and non-negative. The columns of U and V are, respectively, the left and right singular vectors of MAT.

The routine returns the first n singular values and the associated left and right singular vectors (e.g. the left and right singular vectors of MAT in the thin SVD of MAT) or a truncated partial SVD at the user option.

MAT is first reduced to bidiagonal form B with the Ralha-Barlow one-sided bidiagonalization algorithm, see the references (7) and (8). In addition, if m is much larger than n, a preliminary QR factorization is applied to MAT and the one-sided bidiagonalization algorithm is applied to the triangular factor of this QR factorization in order to speedup the computations.

The singular values and vectors of B are then computed by the dqds and inverse iteration methods applied to B, see the references (1), (3) and (4).

The singular vectors of MAT are finally computed by a blocked back-transformation algorithm.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is overwritten as it is used as workspace inside the subroutine.

S (OUTPUT) real(stnd), dimension(:) The singular values of MAT.

The size of S must verify: size(S) = min(m,n) .

U (OUTPUT) real(stnd), dimension(:,:) On exit, the computed first nsvd columns of U, e.g., the left singular vectors associated with the first nsvd largest singular values of MAT.

The left singular vector associated with the singular value S(j) is stored in the j-th column of U.

The shape of V must verify:

- size(U, 1) = size(MAT, 1) = m ,
- size(U, 2) = nsvd <= size(S) = min(m,n) .

V (OUTPUT) real(stnd), dimension(:,:) On exit, the computed first nsvd columns of V, e.g., the right singular vectors associated with the first nsvd largest singular values of MAT.

The right singular vector associated with the singular value S(j) is stored in the j-th column of V.

The shape of V must verify:

- size(V, 1) = size(MAT, 2) = n ,
- size(V, 2) = nsvd <= size(S) = min(m,n) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false indicates successful exit.
- FAILURE = true indicates that the dqds algorithm failed to converge for some singular values or that some singular vectors failed to converge in MAXITER inverse iterations. If the dqds algorithm failed to compute some singular values, the sign of the incorrect singular values in S is set to negative and arguments U and V are filled with a quiet NAN in output of the subroutine.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or into descending order if SORT = 'D' or 'd'. The singular vectors are rearranged accordingly.

The default is to sort the singular values and vectors into descending order.

MAXITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed for computing singular vectors.

By default, 2 inverse iterations are performed for all the singular vectors.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, all the singular vectors are orthogonalized by the Modified Gram-Schmidt or QR algorithm in the inverse iteration algorithm;
- ORTHO=false, the singular vectors are not orthogonalized by the Modified Gram-Schmidt or QR algorithm in the inverse iteration algorithm.

The default is to orthogonalize the singular vectors only for the singular values, which are not well-separated.

BACKWARD_SWEEP (INPUT, OPTIONAL) logical(lgl) On entry, if:

- BACKWARD_SWEEP=true and the singular vectors are orthogonalized by the modified Gram-Schmidt algorithm in the inverse iteration algorithm, a backward sweep of the modified Gram-Schmidt algorithm is also performed;

- BACKWARD_SWEEP=false, a backward sweep is not performed.

The default is not to perform a backward sweep of the modified Gram-Schmidt algorithm.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if:

- SCALING=true, the bidiagonal matrix B is scaled before computing the singular values and vectors;
- SCALING=false, the bidiagonal matrix B is not scaled.

The default is to scale the bidiagonal matrix.

INITVEC (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITVEC=true, Fernando vectors are used to start the inverse iteration process for computing the singular vectors of the bidiagonal matrix BD (e.g. the eigenvectors of the associated Golub-Kahan tridiagonal matrix);
- INITVEC=false, random uniform starting vectors are used.

The default is to use Fernando starting vectors if the singular values are well-separated and the Golub-Kahan form of the input bidiagonal matrix is unreduced, and random uniform starting vectors otherwise.

REORTHO (INPUT, OPTIONAL) logical(lgl) If the optional argument REORTHO is set to true, reorthogonalization is performed (when needed) in the Ralha-Barlow one-sided bidiagonalization of MAT. If this optional logical argument is set to false, reorthogonalization is never performed, which can result in a loss of orthogonality for the left singular vectors of MAT if MAT has a large condition number.

See description of the BD_CMP2 subroutine for further details.

The default is REORTHO = true.

FAILURE_BD (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE_BD = false : indicates that maximum accuracy was obtained in the Ralha-Barlow one-sided bidiagonalization of MAT;
- FAILURE_BD = true : indicates that MAT is nearly singular and some loss of orthogonality can be expected in the Ralha-Barlow bidiagonalization algorithm.

See description of the BD_CMP2 subroutine for further details.

FAILURE_DQDS (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE_DQDS = false : indicates successful exit and the dqds algorithm converged for all the computed singular values to the desired accuracy;
- FAILURE_DQDS = true : indicates that some or all of the singular values failed to converge or were not computed. This is generally caused by unexpectedly inaccurate arithmetic. In that case, U and V are filled with a quiet NAN and the sign of the incorrect singular values in S is set to negative in output of the subroutine.

Further Details

Computing the SVD of a m-by-n matrix MAT, with m>=n, in subroutine SVD_CMP8 consists of three steps:

1) reduction of the rectangular matrix to bidiagonal form B via the Ralha-Barlow one-sided bidiagonalization algorithm, see the references (7) and (8);

2) computation of the SVD of the bidiagonal matrix B with the dqds algorithm for the singular values and inverse iteration for all or the leading singular vectors at the user option;

3) computation of the singular vectors of MAT from those of its bidiagonal form B by a blocked back-transformation algorithm, see the references (1) and (2).

Note that if m is much larger than n, the rectangular matrix is first reduced to upper triangular form by a preliminary QR factorization and the three-step reduction algorithm described above is applied to the resulting triangular factor. The singular vectors of the original rectangular matrix are then obtained from those of the triangular factor in the QR factorization by another blocked back-transformation algorithm.

Concerning the inverse iteration algorithm used here, note that a first estimate of the singular vectors is computed by the Fernando method applied to the tridiagonal Golub-Kahan matrix associated with the bidiagonal matrix B (see the references (5) and (6) for details) for the singular values which are well-separated and if the Golub-Kahan form of the input bidiagonal matrix is unreduced. For the other singular values, a random start is used as a first estimate of the singular vectors as in the standard inverse-iteration algorithm.

The singular vectors of B are then computed or refined using inverse iteration on the tridiagonal Golub-Kahan matrix for all the singular values at one step.

By default, the singular vectors of B are then orthogonalized by the Modified Gram-Schmidt or QR algorithm only if the singular values are not well-separated.

Excepted for the computation of the singular values by the dqds algorithm, all the steps of the SVD algorithm used here (e.g., preliminary QR factorization, reduction to bidiagonal form B, computation of the singular vectors vectors of B, computation of the singular vectors of MAT by blocked back-transformation algorithms are parallelized if OPENMP is used.

For more details, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

(3) **Parlett, B.N., and Marques, O.A., 2000:** An implementation of the dqds algorithm (positive case). Linear Algebra Appl., Volume 309, pp. 217-259.

(4) **Li, S., Gu, M., and Parlett, B. N., 2014:** An Improved DQDS Algorithm. SIAM Journal on Scientific Computing, Volume 36, No 3, C290-C308.

(5) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(6) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

(7) **Barlow, J.L., Bosner, N., and Drmac, Z., 2005:** A new stable bidiagonal reduction algorithm. Linear Algebra Appl., No 397, pp. 35-84.

(8) **Bosner, N., and Barlow, J.L., 2007:** Block and Parallel versions of one-sided bidiagonalization. SIAM J. Matrix Anal. Appl., Volume 29, No 3, pp. 927-953.

6.19.48 subroutine rsvd_cmp (mat, s, leftvec, rightvec, failure, niter, nover, ortho, extd_samp, rng_alg, maxiter, max_francis_steps, perfect_shift, bisect, dqds)

Purpose

RSVD_CMP computes approximations of the nsvd largest singular values and associated left and right singular vectors of a full m-by-n real matrix MAT using randomized power, subspace or block Krylov iterations.

nsvd is the target rank of the partial Singular Value Decomposition (SVD), which is sought, and is equal to the size of the output real vector argument S, i.e., nsvd = size(S).

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT. MAT is not modified by the routine.

S (OUTPUT) real(stnd), dimension(:) On exit, S(:) contains the first top nsvd singular values of MAT. The singular values are given in decreasing order and are positive or zero.

The size of S must verify: size(S) = nsvd <= min(size(MAT,1) , size(MAT,2)) .

LEFTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed nsvd top left singular vectors. The left singular vector associated with the singular value S(j) is stored in the j-th column of LEFTVEC.

The shape of LEFTVEC must verify:

- size(LEFTVEC, 1) = size(MAT, 1) = m ,
- size(LEFTVEC, 2) = size(S) = nsvd .

RIGHTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed nsvd top right singular vectors. The right singular vector associated with the singular value S(j) is stored in the j-th column of RIGHTVEC.

The shape of RIGHTVEC must verify:

- size(RIGHTVEC, 1) = size(MAT, 2) = n ,
- size(RIGHTVEC, 2) = size(S) = nsvd .

FAILURE (OUTPUT, OPTIONAL) logical(lgl) On exit, if the optional logical argument FAILURE is present, a test of the accuracy of the computed singular triplets is performed and in that case:

- FAILURE = false : indicates successful exit;
- FAILURE = true : indicates that some singular values and vectors of MAT failed to converge in NITER iterations.

If FAILURE = true on exit, results are still useful, but some of the approximated singular triplets have a poor accuracy.

NITER (INPUT, OPTIONAL) integer(i4b) The number of randomized power, subspace or block Krylov iterations performed in the subroutine for computing the top nsvd singular triplets. NITER must be positive or null.

By default, 5 randomized power, subspace or block Krylov iterations are performed.

NOVER (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the randomized power, subspace or block Krylov iterations for computing the top nsvd singular triplets.

NOVER must be positive or null and verifies the relationship:

- NOVER + size(S) <= min(size(MAT,1) , size(MAT,2))

and is adjusted if necessary to verify this relationship in all cases.

See Further Details and the cited references for the meaning and usefulness of the oversampling size in the randomized power, subspace or block Krylov iterations.

By default, the oversampling size is set to 10.

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian test matrix in the randomized SVD algorithm.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to RSVD_CMP.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, orthonormalization is carried out between each step of the power or block Krylov iterations, to avoid loss of accuracy due to rounding errors. This means that subspace iterations are used instead of power iterations;
- ORTHO=false, orthonormalization is not performed.

The default is to use orthonormalization, e.g., ORTHO=true.

EXTD_SAMP (INPUT, OPTIONAL) logical(lgl) The optional argument EXTD_SAMP determines if extended sampling (e.g., block Krylov iterations) is used or not for computing the top nsvd singular triplets.

On entry, if:

- EXTD_SAMP=true, block Krylov iterations are used;
- EXTD_SAMP=false, power or subspace iterations are used.

The default is to use power or subspace iterations, e.g., EXTD_SAMP=false.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm, which is used in the last phase of the randomized algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) The optional argument MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g., QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm, which is used in the last phase of the randomized algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

The default is the minimum of nsvd and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) The optional argument PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm, which is used in the last phase of the randomized algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

For a good introduction to randomized linear algebra, see the references (1) and (2).

The randomized power or subspace iteration was proposed in (3; see Algorithm 4.4) to compute an orthonormal matrix whose range approximates the range of MAT. An approximate partial SVD can then be computed using the aforementioned orthonormal matrix, see Algorithm 5.1 in (3).

The randomized block Krylov iterations for computing an approximate partial SVD was proposed in (5; see Algorithm 2). See also the reference (1).

For further details, on randomized linear algebra, computing low-rank matrix approximations and partial SVD using randomized power, subspace or block Krylov iterations, see:

(1) **Martinsson, P.G., 2019:** Randomized methods for matrix computations. arXiv.1607.01649

(2) **Erichson, N.B., Voronin, S., Brunton, S.L., and Kutz, J.N., 2019:** Randomized matrix decompositions using R. arXiv.1608.02148

(3) **Halko, N., Martinsson, P.G., and Tropp, J.A., 2011:** Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions. SIAM Rev., 53, 217-288.

(4) **Gu, M., 2015:** Subspace iteration randomization and singular value problems. SIAM J. Sci. Comput., 37, A1139-A1173.

(5) **Musco, C., and Musco, C., 2015:** Randomized block krylov methods for stronger and faster approximate singular value decomposition. In Proceedings of the 28th International Conference on Neural Information Processing Systems, NIPS 15, pages 1396-1404, Cambridge, MA, USA, 2015. MIT Press.

(6) **Li, H.,Linderman, G.C., Szlam, A., Stanton, K.P., Kluger, Y., and Tygert, M., 2017:** Algorithm 971: An implementation of a randomized algorithm for principal component analysis. ACM Trans. Math. Softw. 43, 3, Article 28 (January 2017).

6.19.49 subroutine rsvd_cmp_fixed_precision (mat, relerr, s, leftvec, rightvec, failure_relerr, failure, niter, blk_size, maxiter_qb, ortho, reortho, niter_qb, rng_alg, maxiter, max_francis_steps, perfect_shift, bisect, dqds)

Purpose

RSVD_CMP_FIXED_PRECISION computes approximations of the top nsvd singular values and associated left and right singular vectors of a full m-by-n real matrix MAT using randomized power or subspace iterations.

nsvd is the target rank of the partial Singular Value Decomposition (SVD), which is sought, and this partial SVD must have an approximation error which fulfills:

||MAT-rSVD||_F <= ||MAT||_F * relerr

, where rSVD is the computed partial SVD approximation, || ||_F is the Frobenius norm and relerr is a prescribed accuracy tolerance for the relative error of the computed partial SVD approximation, specified in the input real argument RELERR.

In other words, nsvd is not known in advance and is determined in the subroutine. This explains why the output real array arguments S, LEFTVEC and RIGHTVEC, which contain the computed singular triplets of the partial SVD on exit, must be declared in the calling program as pointers.

On exit, nsvd is equal to the size of the output real pointer argument S, which contains the computed singular values, i.e., nsvd = size(S) and the relative error in the Frobenius norm of the computed partial SVD approximation is output in argument RELERR.

RSVD_CMP_FIXED_PRECISION searches incrementally the best (e.g., smallest) partial SVD approximation, which fulfills the prescribed accuracy tolerance for the relative error. More precisely, the rank of the partial SVD approximation is increased progressively of BLK_SIZE by BLK_SIZE until the prescribed accuracy tolerance is satisfied and then improved and adjusted precisely by additional subspace iterations (as specified by the optional NITER_QB integer argument) to obtain the smallest partial SVD approximation, which satisfies the prescribed tolerance.

Note that the product of the two integer arguments BLK_SIZE and MAXITER_QB (see below for their meaning), BLK_SIZE*MAXITER_QB, determines the maximum allowable rank of the partial SVD approximation, which is sought. In other words, the subroutine will stop the search for the best (e.g., smallest) partial SVD approximation, which fulfills the requested tolerance, if the rank of this partial SVD approximation exceeds BLK_SIZE*MAXITER_QB. In that case, the subroutine will return the current partial SVD approximation (with a rank less or equal to BLK_SIZE*MAXITER_QB).

In all cases the relative error of the computed partial SVD approximation is output in argument RELERR.

If, finally, the optional logical argument FAILURE_RELERR is used, it will be set to true if the computed partial SVD does not fulfill the requested relative error specified on entry in the argument RELERR and to false otherwise.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

MAT is not modified by the routine.

RELERR (INPUT/OUTPUT) real(stnd) On entry, the requested accuracy tolerance for the relative error of the computed restricted SVD approximation.

The preset RELERR must be greater than 4*epsilon(RELERR), less than one and verifies:

- RELERR >= 2 * sqrt(epsilon(RELERR)/RELERR)

and is forced to be greater than 2*sqrt(epsilon(RELERR)/RELERR) if this is not the case to avoid loss of accuracy in the algorithm. See reference (6) for more details.

On exit, RELERR contains the relative error of the computed partial SVD approximation in the Frobenius norm:

- RELERR = ||MAT-rSVD||_F / ||MAT||_F

S (OUTPUT) real(stnd), dimension(:), pointer On exit, S(:) contains estimates of the first top nsvd singular values of MAT. The singular values are given in decreasing order and are positive or zero.

The statut of the pointer S must not be undefined on entry. If, on entry, the pointer S is already allocated, it will be first deallocated and then reallocated with the correct size.

On exit, the size of the pointer S will verify:

- size(S) = nsvd <= min(size(MAT,1) , size(MAT,2)) .

LEFTVEC (OUTPUT) real(stnd), dimension(:,:), pointer On exit, the computed nsvd top left singular vectors. The left singular vector associated with the singular value S(j) is stored in the j-th column of LEFTVEC.

The statut of the pointer LEFTVEC must not be undefined on entry. If, on entry, the pointer LEFTVEC is already allocated, it will be first deallocated and then reallocated with the correct shape.

On exit, the shape of the pointer LEFTVEC will verify:

- size(LEFTVEC, 1) = size(MAT, 1) = m ,
- size(LEFTVEC, 2) = size(S) = nsvd .

RIGHTVEC (OUTPUT) real(stnd), dimension(:,:), pointer On exit, the computed nsvd top right singular vectors. The right singular vector associated with the singular value S(j) is stored in the j-th column of RIGHTVEC.

The statut of the pointer RIGHTVEC must not be undefined on entry. If, on entry, the pointer RIGHTVEC is already allocated, it will be first deallocated and then reallocated with the correct shape.

On exit, the shape of the pointer RIGHTVEC will verify:

- size(RIGHTVEC, 1) = size(MAT, 2) = n ,
- size(RIGHTVEC, 2) = size(S) = nsvd .

FAILURE_RELERR (OUTPUT, OPTIONAL) logical(lgl) On exit, if the optional logical argument FAILURE_RELERR is present, it is set on exit as follows:

- FAILURE_RELERR = false : indicates successful exit and the computed partial SVD fulfills the requested relative error specified on entry in the argument RELERR,
- FAILURE_RELERR = true : indicates that the computed partial SVD has a relative error larger than the requested relative error. This means that the requested accuracy tolerance for the relative error is too small (i.e., RELERR < 2 * sqrt(epsilon(RELERR)/RELERR) or that the input parameters BLK_SIZE and/or MAXITER_QB have a too small value, given the distribution of the singular values of MAT, and must be increased to fullfill the preset accuracy tolerance for the relative error of the partial SVD approximation.

FAILURE (OUTPUT, OPTIONAL) logical(lgl) On exit, if the optional logical argument FAILURE is present, a test of the accuracy of the computed singular triplets is performed and in that case:

- FAILURE = false : indicates successful exit;
- FAILURE = true : indicates that some singular values and vectors of MAT failed to converge in NITER and NITER_QB power and subspace iterations.

If FAILURE = true on exit, results are still useful, but some of the approximated singular triplets have a poor accuracy.

NITER (INPUT, OPTIONAL) integer(i4b) The number of randomized power or subspace iterations performed in the first phase of the randomized algorithm for computing the preliminary QB factorization.

NITER must be positive or null.

By default, 1 randomized power or subspace iteration is performed.

BLK_SIZE (INPUT, OPTIONAL) integer(i4b) On entry, the block size used in the randomized QB factorization, which is used in the first phase of the randomized SVD algorithm.

BLK_SIZE must be greater or equal to one and less than min(m,n) and must be set to a much smaller value than min(m,n) usually, depending also on the architecture of the computer.

By default, BLK_SIZE is set to min(10, min(m,n)).

MAXITER_QB (INPUT, OPTIONAL) integer(i4b) MAXITER_QB controls the maximum number of allowed iterations in the randomized QB algorithm, which is used in the first phase of the randomized SVD algorithm.

MAXITER_QB must be set greater or equal to one and less than int(min(m,n)/BLK_SIZE).

By default, MAXITER_QB is set to max(1, int(min(m,n)/(4*BLK_SIZE))).

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, orthonormalization is carried out between each step of the power iterations, to avoid loss of accuracy due to rounding errors. This means that subspace iterations are used instead of power iterations,
- ORTHO=false, orthonormalization is not performed.

The default is to use orthonormalization, e.g., ORTHO=true.

REORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- REORTHO=true, a reorthogonalization step is performed to avoid the loss of orthogonality in the Gram-Schmidt procedure, which is used in the randomized QB factorization;
- REORTHO=false, a reorthogonalization step is not performed in the Gram-Schmidt procedure.

The default is to use a reorthogonalization step, e.g., REORTHO=true.

NITER_QB (INPUT, OPTIONAL) integer(i4b) The number of subspace iterations performed in the last phase of the QB algorithm for improving the QB factorization and computes the top nsvd singular triplets of MAT.

NITER_QB must be greater or equal to 0.

By default, 2 final subspace iterations are performed.

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian test matrix in the randomized SVD algorithm.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to RSVD_CMP_FIXED_PRECISION.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm, which is used in the last phase of the randomized algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) The optional argument MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g., QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm, which is used in the last phase of the randomized algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

The default is the minimum of nsvd and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) The optional argument PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm, which is used in the last phase of the randomized algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

For a good introduction to randomized linear algebra , see the references (1) and (2).

The randomized power or subspace iteration was proposed in (3; see Algorithm 4.4) to compute an orthonormal matrix whose range approximates the range of MAT. An approximate partial SVD can then be computed using the aforementioned orthonormal matrix, see Algorithm 5.1 in (3).

Usually, the problem of low-rank matrix approximation falls into two categories:

- the fixed-rank problem, where the rank parameter nsvd is given;
- the fixed-precision problem, where we seek a partial SVD factorization, rSVD, as small as possible such that

  ||MAT-rSVD||_F <= eps

  , where eps is a given accuracy tolerance.

RSVD_CMP_FIXED_PRECISION is dedicated to solve the fixed-precision problem. The fixed-rank problem can be solved by subroutine RSVD_CMP.

RSVD_CMP_FIXED_PRECISION uses an improved version of the "randQB_FP" algorithm described in the reference (6) to solve the fixed-precision problem.

For further details, on randomized linear algebra, computing low-rank matrix approximations, partial SVD using randomized power or subspace iterations or solving the fixed-precision problem, see:

(1) **Martinsson, P.G., 2019:** Randomized methods for matrix computations. arXiv.1607.01649

(2) **Erichson, N.B., Voronin, S., Brunton, S.L., and Kutz, J.N., 2019:** Randomized matrix decompositions using R. arXiv.1608.02148

(3) **Halko, N., Martinsson, P.G., and Tropp, J.A., 2011:** Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions. SIAM Rev., 53, 217-288.

(4) **Gu, M., 2015:** Subspace iteration randomization and singular value problems. SIAM J. Sci. Comput., 37, A1139-A1173.

(5) **Martinsson, P.G., and Voronin, S., 2016:** A randomized blocked algorithm for efficiently computing rank-revealing factorizations of matrices. SIAM J. Sci. Comput., 38:5, S485-S507.

(6) **Yu, W., Gu, Y., and Li, Y., 2018:** Efficient randomized algorithms for the fixed-precision low-rank matrix approximation. SIAM J. Mat. Ana. Appl., 39:3, 1339-1359.

6.19.50 subroutine reig_pos_cmp (mat, eigval, eigvec, failure, niter, nover, ortho, extd_samp, use_nystrom, rng_alg, maxiter, max_francis_steps, perfect_shift, bisect, dqds)

Purpose

REIG_POS_CMP computes approximations of the neig largest eigenvalues and associated eigenvectors of a full n-by-n real symmetric positive semi-definite matrix MAT using randomized power, subspace or block Krylov iterations and, at the user option, the Nystrom method (see below for details).

neig is the target rank of the partial EigenValue Decomposition (EVD), which is sought, and is equal to the size of the output real vector argument EIGVAL, i.e., neig = size(EIGVAL).

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the n-by-n symmetric positive semi-definite matrix MAT.

MAT is not modified by the routine.

EIGVAL (OUTPUT) real(stnd), dimension(:) On exit, EIGVAL(:) contains the first top neig eigenvalues of MAT. The eigenvalues are given in decreasing order of magnitude.

The size of EIGVAL must verify:

- size(EIGVAL) = neig <= size(MAT, 1) = size(MAT, 2) = n.

EIGVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed neig top eigenvectors. The eigenvector associated with the eigenvalue EIGVAL(j) is stored in the j-th column of EIGVEC.

The shape of EIGVEC must verify:

- size(EIGVEC, 1) = size(MAT, 1) = size(MAT, 2) = n,
- size(EIGVEC, 2) = size(EIGVEC) = neig .

If FAILURE = true on exit, results are still useful, but some of the approximated eigen couplets have a poor accuracy.

FAILURE (OUTPUT, OPTIONAL) logical(lgl) On exit, if the optional logical argument FAILURE is present, a test of the accuracy of the computed partial EVD is performed and in that case:

- FAILURE = false : indicates successful exit;
- FAILURE = true : indicates that some of the computed eigenvalues and eigenvectors of MAT failed to converge in NITER iterations.

If FAILURE = true on exit, results are still useful, but some of the approximated eigen couplets have a poor accuracy.

NITER (INPUT, OPTIONAL) integer(i4b) The number of randomized power, subspace or block Krylov iterations performed in the subroutine for computing the top neig eigen triplets. NITER must be positive or null.

By default, 10 randomized power, subspace or block Krylov iterations are performed.

NOVER (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the randomized power, subspace or block Krylov iterations for computing the top neig eigen triplets.

NOVER must be positive or null and verifies the relationship:

- NOVER + size(EIGVAL) <= size(MAT, 1) = size(MAT, 2) = n

and is adjusted if necessary to verify this relationship in all cases.

See Further Details and the cited references for the meaning and usefulness of the oversampling size in the randomized power, subspace or block Krylov iterations.

By default, the oversampling size is set to 10.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, orthonormalization is carried out between each step of the power iterations to avoid loss of accuracy due to rounding errors. This means that subspace iterations are used instead of power iterations;
- ORTHO=false, orthonormalization is not performed.

The default is to use orthonormalization, e.g., ORTHO=true.

EXTD_SAMP (INPUT, OPTIONAL) logical(lgl) The optional argument EXTD_SAMP determines if extended sampling (e.g., block Krylov iterations) is used or not for computing the top neig eigen triplets.

On entry, if:

- EXTD_SAMP=true, block Krylov iterations are used;
- EXTD_SAMP=false, power or subspace iterations are used.

The default is to use power or subspace iterations, e.g., EXTD_SAMP=false.

USE_NYSTROM (INPUT, OPTIONAL) logical(lgl) If the optional argument USE_NYSTROM is used and is set to:

- true, the last step of the randomized algorithm is performed with the Nystrom method and an SVD decomposition;
- false, an EVD decomposition is used in the final step of the randomized algorithm.

The default is to use the Nystrom method, e.g., USE_NYSTROM=true.

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian test matrix in the randomized EVD algorithm.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to REIG_POS_CMP.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm or in the QR phase of the EVD algorithm, which are used in the last phase of the randomized algorithm.

See description of suboutines SVD_CMP and EIG_CMP for further details about this optional argument.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) The optional argument MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm, which is used in the last phase of the randomized algorithm if the Nystrom method is used.

See description of suboutine SVD_CMP for further details about this optional argument.

The default is the minimum of neig and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) The optional argument PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm, which is used in the last phase of the randomized EVD algorithm if the Nystrom method is used.

See description of suboutine SVD_CMP for further details about this optional argument.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false or if the Nystrom method is not used.

If PERFECT_SHIFT and BISECT are both set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed EVD decomposition at the expense of a slightly slower execution time if the Nystrom method is used.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm if the Nystrom method is used and PERFECT_SHIFT is equal to true.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

For a good introduction to randomized linear algebra, see the references (1) and (2).

The randomized subspace iteration was proposed in (3; see Algorithm 4.4) to compute an orthonormal matrix whose range approximates the range of MAT. An approximate partial spectral decomposition can then be computed using the aforementioned orthonormal matrix, see Algorithm 5.3 in (3). Moreover, if the input matrix is positive semi-definite, an improved randomized algorithm exists, the Nystrom method, see Algorithm 5.5 in (3) and also references (1) and (5).

The Nystrom method will be selected in REIG_POS_CMP if the USE_NYSTROM argument is used with the value true (this is the default), otherwise the standard EVD algorithm will be used in the last step of the randomized algorithm. The Nystrom method provides more accurate results for positive (semi-)definite matrices.

The randomized block Krylov iterations for computing an approximate partial EVD was proposed in (4; see Algorithm 2). See also the reference (1).

For further details on randomized linear algebra, computing a partial EVD decomposition using randomized power, subspace or block Krylov iterations, or the Nystrom method, see:

(1) **Martinsson, P.G., 2019:** Randomized methods for matrix computations. arXiv.1607.01649

(2) **Erichson, N.B., Voronin, S., Brunton, S.L., and Kutz, J.N., 2019:** Randomized matrix decompositions using R. arXiv.1608.02148

(3) **Halko, N., Martinsson, P.G., and Tropp, J.A., 2011:** Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions. SIAM Rev., 53, 217-288.

(4) **Musco, C., and Musco, C., 2015:** Randomized block krylov methods for stronger and faster approximate singular value decomposition. In Proceedings of the 28th International Conference on Neural Information Processing Systems, NIPS 15, pages 1396-1404, Cambridge, MA, USA, 2015. MIT Press.

(5) **Li, H.,Linderman, G.C., Szlam, A., Stanton, K.P., Kluger, Y., and Tygert, M., 2017:** Algorithm 971: An implementation of a randomized algorithm for principal component analysis. ACM Trans. Math. Softw. 43, 3, Article 28 (January 2017).

6.19.51 subroutine rqr_svd_cmp (mat, s, failure, v, random_qr, truncated_qr, rng_alg, blk_size, nover, nover_svd, maxiter, max_francis_steps, perfect_shift, bisect, dqds)

Purpose

RQR_SVD_CMP computes approximations of the nsvd largest singular values and associated left and right singular vectors of a full m-by-n real matrix MAT using a three-step procedure, which can be termed a QR-SVD algorithm:

- first, a partial (or complete) QR factorization with column pivoting of MAT is computed;
- in a second step, a Singular Value Decomposition (SVD) of the (permuted) upper triangular or trapezoidal (e.g., if n>m) factor, R, of this QR decomposition is computed. The singular values and right singular vectors of this SVD of R are also estimates of the singular values and right singular vectors of MAT;
- Estimates of the associated left singular vectors of MAT are then obtained by pre-multiplying the left singular of R by the orthogonal matrix Q in the initial QR decomposition of MAT (or its first k columns if the QR factorization is only partial).

By default, a standard deterministic BLAS2 QR factorization with column pivoting is used in the first phase of the QR-SVD algorithm. However, if the optional logical argument RANDOM_QR is used with the value true, an alternate fast randomized partial QR factorization is used in the first phase of the QR-SVD algorithm.

Furthermore if, in addition, the optional logical argument TRUNCATED_QR is used with the value true, an even faster (but less accurate) randomized partial and truncated QR factorization is used in the first phase of the QR-SVD algorithm.

nsvd is the target rank of the partial SVD, which is sought, and is equal to the size of the output real vector argument S, i.e., nsvd = size(S). If, nsvd = min(size(MAT,1) , size(MAT,2)), a full SVD of MAT is obtained with the same or higher accuracy than subroutines SVD_CMP or SVD_CMP2 if the optional logical argument RANDOM_QR is not used (or is set to false).

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, the top nsvd left singular vectors are stored in the first nsvd columns of MAT. The left singular vector associated with the singular value S(j) is stored in the j-th column of MAT. The other part of MAT is used as workspace in the algorithm and is destroyed on exit.

S (OUTPUT) real(stnd), dimension(:) On exit, S(:) contains the first top nsvd singular values of MAT. The singular values are given in decreasing order and are positive or zero.

The size of S must verify:

- size(S) = nsvd <= min(size(MAT,1) , size(MAT,2)) .

FAILURE (OUTPUT) logical(lgl) On exit, if:

- FAILURE = false : indicates successful exit in the SVD of the triangular factor R of the QR decomposition of MAT.
- FAILURE = true : indicates that the SVD algorithm did not converge and that full accuracy was not attained in the bidiagonal SVD of an intermediate bidiagonal form B of the triangular factor R in the QR decomposition of MAT.

If on entry, RANDOM_QR=true and TRUNCATED_QR=true, a test of the accuracy of the randomized partial and truncated QR factorization used in the first phase is also performed. In that case:

- FAILURE = false : indicates also that this randomized partial and truncated QR factorization seems accurate.
- FAILURE = true : indicates that this randomized partial and truncated QR factorization is not accurate.

V (OUTPUT) real(stnd), dimension(:,:) On exit, the computed top nsvd right singular vectors of MAT. The right singular vector associated with the singular value S(j) is stored in the j-th column of V.

The shape of V must verify:

- size(V, 1) = size(MAT, 2) = n ,
- size(V, 2) = size(S) = nsvd .

RANDOM_QR (INPUT, OPTIONAL) logical(lgl) On entry, if RANDOM_QR is used with the value true, a fast randomized partial QR factorization is used in the first phase of the QR-SVD algorithm.

By default, RANDOM_QR = false, i.e., a standard deterministic (partial) QR factorization with column pivoting is used in the first phase of the QR-SVD algorithm.

TRUNCATED_QR (INPUT, OPTIONAL) logical(lgl) On entry, if TRUNCATED_QR is used with the value true in addition to RANDOM_QR also set to true, a very fast (but less accurate) randomized partial and truncated QR factorization is used in the first phase of the QR-SVD algorithm.

By default, TRUNCATED_QR = false, i.e., a "standard" randomized (partial) QR factorization with column pivoting is used in the first phase of the QR-SVD algorithm if RANDOM_QR = true.

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian matrix in the randomized (partial) QR phase of the QR-SVD algorithm if RANDOM_QR = true.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to RQR_SVD_CMP.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

BLK_SIZE (INPUT, OPTIONAL) integer(i4b) On entry, the block size used in the randomized partial QR phase of the QR-SVD algorithm if RANDOM_QR = true (and TRUNCATED_QR = false).

BLK_SIZE must be greater or equal to one and less than min(m,n) and must be set to a much smaller value than min(m,n) usually, depending also on the architecture of the computer.

See Further Details and the cited references for the meaning of the block size in the randomized (partial) QR algorithm.

By default, BLK_SIZE is set to min(BLKSZ_QR, min(m,n)), where parameter BLKSZ_QR is the default block size for QR related algorithms specified in module Select_Parameters.

NOVER (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the randomized partial QR phase of the QR-SVD algorithm if RANDOM_QR = true.

NOVER must be positive or null and verify the relationships:

- NOVER + BLK_SIZE <= size(MAT, 1) if TRUNCATED_QR = false;
- NOVER + NOVER_SVD + size(S) <= size(MAT, 1) if TRUNCATED_QR = true.

and is adjusted if necessary to verify these relationships in all cases.

See Further Details and the cited references for the meaning and usefulness of the oversampling size in the randomized (partial) QR algorithms.

By default, the oversampling size is set to:

- 10 if TRUNCATED_QR = false;
- max((NOVER_SVD+size(S))/2_i4b, 10) if TRUNCATED_QR = true.

NOVER_SVD (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the SVD phase of the QR-SVD algorithm for computing the top nsvd singular triplets.

NOVER_SVD must be positive or null and verify the relationship:

- NOVER_SVD + size(S) <= min(size(MAT,1) , size(MAT,2))

and is adjusted if necessary to verify this relationship in all cases.

See Further Details and the cited references for the meaning and usefulness of the oversampling size in the SVD phase of the QR-SVD algorithm.

By default, the oversampling size in the SVD phase is set to 10.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm, which is used in the last step of the QR-SVD algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) The optional argument MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g., QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm, which is used in the last step of the QR-SVD algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

The default is the minimum of nsvd and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) The optional argument PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm, which is used in the last step of the QR-SVD algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed QR-SVD decomposition at the expense of a slightly slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

The standard deterministic BLAS2 algorithm for computing a QR factorization with column pivoting is described in the reference (1). The randomized partial QR factorization with column pivoting used if the optional logical argument RANDOM_QR is present with the value true is described in the references (3), (4), (5) and (6). Finally, the randomized partial and truncated QR factorization with column pivoting used if both the optional logical arguments RANDOM_QR and TRUNCATED_QR are present with the value true is described in the reference (7). This algorithm is the fastest, but is less accurate than the randomized partial QR factorization with column pivoting described in the references (3), (4), (5) and (6).

For further details, on computing low-rank matrix approximations from QR factorizations with column pivoting, the QR-SVD or randomized QR algorithms, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Chan, T.F., and Hansen, P.C., 1992:** Some applications of the rank revealing QR factorization. SIAM J. Sci. Statist. Comput., Volume 13, 727-741.

(3) **Duersch, J.A., and Gu, M., 2017:** Randomized QR with column pivoting. SIAM J. Sci. Comput., Volume 39, C263-C291.

(4) **Martinsson, P.G., Quintana-Orti, G., Heavner, N., and Van de Geijn, R., 2017:** Householder QR factorization with randomization for column pivoting (HQRRP). SIAM J. Sci. Comput., Volume 39, C96-C115.

(5) **Xiao, J., Gu, M., and Langou, J., 2017:** Fast parallel randomized QR with column pivoting algorithms for reliable low-rank matrix approximations. IEEE 24th International Conference on High Performance Computing (HiPC), IEEE, 2017, 233-242.

(6) **Duersch, J.A., and Gu, M., 2020:** Randomized projection for rank-revealing matrix factorizations and low-rank approximations. SIAM Review, Volume 62, Issue 3, 661-682.

(7) **Mary, T., Yamazaki, I., Kurzak, J., Luszczek, P., Tomov, S., and Dongarra, J., 2015:** Performance of Random Sampling for Computing Low-rank Approximations of a Dense Matrix on GPUs. International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15).

6.19.52 subroutine rqr_svd_cmp_fixed_precision (mat, relerr, s, failure, v, random_qr, rng_alg, blk_size, nover, maxiter, max_francis_steps, perfect_shift, bisect, dqds)

Purpose

RQR_SVD_CMP_FIXED_PRECISION computes approximations of the nsvd largest singular values and associated left and right singular vectors of a full m-by-n real matrix MAT using a three-step procedure, which can be termed a QR-SVD algorithm:

- first, a partial QR factorization with column pivoting of MAT is computed;
- in a second step, a Singular Value Decomposition (SVD) of the (permuted) upper triangular or trapezoidal (e.g., if n>m) factor, R, of this QR decomposition is computed. The singular values and right singular vectors of this SVD of R are also estimates of the singular values and right singular vectors of MAT.
- in a final step, estimates of the associated left singular vectors of MAT are obtained by pre-multiplying the left singular of R by the orthogonal matrix Q of the initial QR decomposition (or its first k columns if the QR factorization is only partial).

By default, a standard deterministic BLAS2 QR factorization with column pivoting is used in the first phase of the QR-SVD algorithm. However, if the optional logical argument RANDOM_QR is used with the value true, an alternate fast randomized partial QR factorization is used in the first phase of the QR-SVD algorithm.

nsvd is the target rank of the partial Singular Value Decomposition (SVD), which is sought, and this partial SVD must have an approximation error which fulfills:

||MAT-rSVD||_F <= ||MAT||_F * relerr

, where rSVD is the computed partial SVD approximation, || ||_F is the Frobenius norm and relerr is a prescribed accuracy tolerance for the relative error of the computed partial SVD approximation, which is specified in the input real argument RELERR.

In other words, nsvd is not known in advance and is determined in the subroutine. This explains why the output real array arguments S and V, which contain the computed singular values and associated right singular vectors of MAT on exit, must be declared in the calling program as pointers.

On exit, nsvd is equal to the size of the output real pointer argument S, which contains the computed singular values, i.e., nsvd = size(S) and the relative error in the Frobenius norm of the computed partial SVD approximation is output in argument RELERR.

RQR_SVD_CMP_FIXED_PRECISION searches incrementally the best (e.g., smallest) partial SVD approximation, which fulfills the prescribed accuracy tolerance for the relative error. More precisely, the rank of the partial SVD approximation is increased progressively until the prescribed accuracy tolerance is satisfied and then improved and adjusted precisely in a final step to obtain the smallest partial SVD approximation, which satisfies the prescribed tolerance.

In all cases the relative error of the computed partial SVD approximation is output in argument RELERR.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, the top nsvd left singular vectors are stored in the first nsvd columns of MAT. The left singular vector associated with the singular value S(j) is stored in the j-th column of MAT. The other part of MAT is used as workspace in the algorithm and is destroyed on exit.

RELERR (INPUT/OUTPUT) real(stnd) On entry, the requested accuracy tolerance for the relative error of the computed partial SVD approximation.

The preset value for RELERR must be greater than 4*epsilon(RELERR) and less than one.

On exit, RELERR contains the relative error of the computed partial SVD approximation in the Frobenius norm:

- RELERR = ||MAT-rSVD||_F / ||MAT||_F

S (OUTPUT) real(stnd), dimension(:), pointer On exit, S(:) contains estimates of the first top nsvd singular values of MAT. The singular values are given in decreasing order and are positive or zero.

The statut of the pointer S must not be undefined on entry. If, on entry, the pointer S is already allocated, it will be first deallocated and then reallocated with the correct size.

On exit, the size of the pointer S will verify:

- size(S) = nsvd <= min(size(MAT,1) , size(MAT,2)) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit in the SVD of the triangular factor R of the QR decomposition of MAT.
- FAILURE = true : indicates that the SVD algorithm did not converge and that full accuracy was not attained in the bidiagonal SVD of an intermediate bidiagonal form B of the triangular factor R of the QR decomposition of MAT.

V (OUTPUT) real(stnd), dimension(:,:), pointer On exit, the computed nsvd top right singular vectors. The right singular vector associated with the singular value S(j) is stored in the j-th column of V.

The statut of the pointer V must not be undefined on entry. If, on entry, the pointer V is already allocated, it will be first deallocated and then reallocated with the correct shape.

On exit, the shape of the pointer V will verify:

- size(V, 1) = size(MAT, 2) = n ,
- size(V, 2) = size(S) = nsvd .

RANDOM_QR (INPUT, OPTIONAL) logical(lgl) On entry, if RANDOM_QR is used with the value true, a fast randomized partial QR factorization is used in the first phase of the QR-SVD algorithm.

By default, RANDOM_QR = false, i.e., a standard deterministic (partial) QR factorization with column pivoting is used in the first phase of the QR-SVD algorithm.

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian matrix in the randomized (partial) QR phase of the QR-SVD algorithm if RANDOM_QR = true.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to RQR_SVD_CMP_FIXED_PRECISION.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

BLK_SIZE (INPUT, OPTIONAL) integer(i4b) On entry, the block size used in the randomized partial QR phase of the QR-SVD algorithm if RANDOM_QR = true.

BLK_SIZE must be greater or equal to one and less than min(m,n) and must be set to a much smaller value than min(m,n) usually, depending also on the architecture of the computer.

See Further Details and the cited references for the meaning of the block size in the randomized (partial) QR algorithm.

By default, BLK_SIZE is set to min(BLKSZ_QR, min(m,n)), where parameter BLKSZ_QR is the default block size for QR related algorithms specified in module Select_Parameters.

NOVER (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the randomized partial QR phase of the QR-SVD algorithm if RANDOM_QR = true.

NOVER must be positive or null and verify the relationship:

- NOVER + BLK_SIZE <= size(MAT, 1)

and is adjusted if necessary to verify this relationship in all cases.

See Further Details and the cited references for the meaning and usefulness of the oversampling size in the randomized (partial) QR algorithm.

By default, the oversampling size is set to 10.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm, which is used in the last step of the QR-SVD algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) The optional argument MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g., QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm, which is used in the last step of the QR-SVD algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

The default is the minimum of nsvd and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) The optional argument PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm, which is used in the last step of the QR-SVD algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed QR-SVD decomposition at the expense of a slightly slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

Usually, the problem of low-rank matrix approximation falls into two categories:

- the fixed-rank problem, where the rank parameter nsvd is given;

- the fixed-precision problem, where we seek a partial SVD factorization, rSVD, as small as possible such that

  ||MAT-rSVD||_F <= eps

  , where eps is a given accuracy tolerance.

RQR_SVD_CMP_FIXED_PRECISION is dedicated to solve the fixed-precision problem. The fixed-rank problem can be solved by subroutine RQR_SVD_CMP.

The standard deterministic BLAS2 algorithm for computing a QR factorization with column pivoting is described in the reference (1). The randomized partial QR algorithm with column pivoting used if the optional logical argument RANDOM_QR is present with the value true is described in the references (3), (4), (5) and (6).

For further details, on computing low-rank matrix approximations from QR factorizations with column pivoting, the QR-SVD or randomized QR algorithms, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Chan, T.F., and Hansen, P.C., 1992:** Some applications of the rank revealing QR factorization. SIAM J. Sci. Statist. Comput., Volume 13, 727-741.

(3) **Duersch, J.A., and Gu, M., 2017:** Randomized QR with column pivoting. SIAM J. Sci. Comput., Volume 39, C263-C291.

(4) **Martinsson, P.G., Quintana-Orti, G., Heavner, N., and Van de Geijn, R., 2017:** Householder QR factorization with randomization for column pivoting (HQRRP). SIAM J. Sci. Comput., Volume 39, C96-C115.

(5) **Xiao, J., Gu, M., and Langou, J., 2017:** Fast parallel randomized QR with column pivoting algorithms for reliable low-rank matrix approximations. IEEE 24th International Conference on High Performance Computing (HiPC), IEEE, 2017, 233-242.

(6) **Duersch, J.A., and Gu, M., 2020:** Randomized projection for rank-revealing matrix factorizations and low-rank approximations. SIAM Review, Volume 62, Issue 3, 661-682.

6.19.53 subroutine rqlp_svd_cmp (mat, s, leftvec, rightvec, failure, niter, random_qr, truncated_qr, rng_alg, blk_size, nover, nover_svd, maxiter, max_francis_steps, perfect_shift, bisect, dqds)

Purpose

RQLP_SVD_CMP computes approximations of the nsvd largest singular values and associated left and right singular vectors of a full m-by-n real matrix MAT using a four-step procedure, which can be termed a QLP-SVD algorithm:

- First, a partial (or complete) QLP factorization of MAT is computed as

  MAT = Q * L * P

  , where Q is a m-by-m (or m-by-k if the factorization is only partial) orthogonal matrix, P is an n-by-n (or k-by-n if the factorization is partial) orthogonal matrix and L is a lower m-by-n (or k-by-k if the factorization is partial) triangular matrix.

- In a second step, the matrix product MAT * P' is computed and, at the user option, a number of QR-QL iterations are performed (this is equivalent to subspace iterations) to improve the estimates of the principal row and columns subspaces of MAT.

- In a third step, a Singular Value Decomposition (SVD) of the matrix product MAT * P' is computed. The singular values and left singular vectors of this SVD are also estimates of the singular values and left singular vectors of MAT.
- In a final step, estimates of the associated right singular vectors of MAT are obtained by pre-multiplying P' by the right singular vectors in the SVD of this matrix product.

If the optional logical argument RANDOM_QR is used with the value true, a fast randomized (partial) QLP factorization is used in the first phase of the QLP-SVD algorithm. Furthermore if, in addition, the optional logical argument TRUNCATED_QR is used with the value true, an even faster (but slightly less accurate) randomized partial and truncated QLP factorization will be used in the first phase of the QLP-SVD algorithm.

nsvd is the target rank of the partial SVD, which is sought, and is equal to the size of the output real vector argument S, i.e., nsvd = size(S). If, nsvd = min(size(MAT,1) , size(MAT,2)), a full SVD of MAT is obtained with the same or higher accuracy than subroutines SVD_CMP or SVD_CMP2.

See Further Details and the cited references for more information.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is not modified by the routine.

S (OUTPUT) real(stnd), dimension(:) On exit, S(:) contains the first top nsvd singular values of MAT. The singular values are given in decreasing order and are positive or zero.

The size of S must verify:

- size(S) = nsvd <= min(size(MAT,1) , size(MAT,2)) .

LEFTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed top nsvd left singular vectors of MAT. The left singular vector associated with the singular value S(j) is stored in the j-th column of LEFTVEC.

The shape of LEFTVEC must verify:

- size(LEFTVEC, 1) = size(MAT, 1) = m ,
- size(LEFTVEC, 2) = size(S) = nsvd .

RIGHTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed top nsvd right singular vectors of MAT. The right singular vector associated with the singular value S(j) is stored in the j-th column of RIGHTVEC.

The shape of RIGHTVEC must verify:

- size(RIGHTVEC, 1) = size(MAT, 2) = n ,
- size(RIGHTVEC, 2) = size(S) = nsvd .

FAILURE (OUTPUT, OPTIONAL) logical(lgl) On exit, if the optional logical argument FAILURE is present, a test of the accuracy of the computed singular triplets is performed and in that case:

- FAILURE = false : indicates successful exit;
- FAILURE = true : indicates that some singular values and vectors of MAT failed to converge in NITER QR-QL iterations.

If FAILURE = true on exit, results are still useful, but some of the approximated singular triplets have a poor accuracy.

NITER (INPUT, OPTIONAL) integer(i4b) The number of subspace iterations performed in the subroutine after the initial QLP factorization and first subspace projection for computing the top nsvd singular triplets.

NITER must be positive or null.

By default, no subspace iterations are performed after the initial QLP factorization and first subspace projection.

RANDOM_QR (INPUT, OPTIONAL) logical(lgl) On entry, if RANDOM_QR is used with the value true, a fast randomized partial QLP factorization is used in the first phase of the QLP-SVD algorithm.

By default, RANDOM_QR = false, i.e., a standard (partial) deterministic QLP factorization is used in the first phase of the QLP-SVD algorithm.

TRUNCATED_QR (INPUT, OPTIONAL) logical(lgl) On entry, if TRUNCATED_QR is used with the value true in addition to RANDOM_QR also set to true, a very fast (but slightly less accurate) randomized partial QLP factorization is used in the first phase of the QLP-SVD algorithm.

By default, TRUNCATED_QR = false, i.e., a "standard" randomized (partial) QLP factorization is used in the first phase of the QLP-SVD algorithm if RANDOM_QR = true.

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian matrix in the randomized (partial) QLP phase of the QLP-SVD algorithm if RANDOM_QR = true.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to RQLP_SVD_CMP.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

BLK_SIZE (INPUT, OPTIONAL) integer(i4b) On entry, the block size used in the randomized partial QLP phase of the QLP-SVD algorithm if RANDOM_QR = true (and TRUNCATED_QR = false).

BLK_SIZE must be greater or equal to one and less than min(m,n) and must be set to a much smaller value than min(m,n) usually, depending also on the architecture of the computer.

See Further Details and the cited references for the meaning of the block size in the randomized (partial) QR or QLP algorithm.

By default, BLK_SIZE is set to min(BLKSZ_QR, min(m,n)), where parameter BLKSZ_QR is the default block size for QR related algorithms specified in module Select_Parameters.

NOVER (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the randomized partial QLP phase of the QLP-SVD algorithm if RANDOM_QR = true.

NOVER must be positive or null and verify the relationships:

- NOVER + BLK_SIZE <= size(MAT, 1) if TRUNCATED_QR = false;
- NOVER + NOVER_SVD + size(S) <= size(MAT, 1) if TRUNCATED_QR = true.

and is adjusted if necessary to verify these relationships in all cases.

See Further Details and the cited references for the meaning and usefulness of the oversampling size in the partial randomized QR and QLP algorithms.

By default, the oversampling size is set to:

- 10 if TRUNCATED_QR = false;
- max((NOVER_SVD+size(S))/2_i4b, 10) if TRUNCATED_QR = true.

NOVER_SVD (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the SVD phase of the QLP-SVD algorithm for computing the top nsvd singular triplets.

NOVER_SVD must be positive or null and verify the relationship:

- NOVER_SVD + size(S) <= min(size(MAT,1) , size(MAT,2))

and is adjusted if necessary to verify this relationship in all cases.

See Further Details and the cited references for the meaning and usefulness of the oversampling size in the SVD phase of the QLP-SVD algorithm.

By default, the oversampling size is set to 10.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm, which is used in the last step of the QLP-SVD algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) The optional argument MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g., QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm, which is used in the last step of the QLP-SVD algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

The default is the minimum of nsvd and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) The optional argument PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm, which is used in the last step of the QLP-SVD algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed QLP-SVD decomposition at the expense of a slightly slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

The QLP-SVD algorithm implemented in RQLP_SVD_CMP subroutine is a variation of the TXUV algorithm described in the references (2), (5) and (6).

For further details, on computing low-rank matrix approximations with a QLP factorization, the QLP-SVD (e.g., TXUV) algorithm or randomized (partial) QLP and QR factorizations, see:

(1) **Stewart, G.W., 1999:** The QLP approximation to the singular value decomposition. SIAM J. Sci. Comput., Volume 20, 1336-1348.

(2) **Duersch, J.A., and Gu, M., 2017:** Randomized QR with column pivoting. SIAM J. Sci. Comput., Volume 39, C263-C291.

(3) **Martinsson, P.G., Quintana-Orti, G., Heavner, N., and Van de Geijn, R., 2017:** Householder QR factorization with randomization for column pivoting (HQRRP). SIAM J. Sci. Comput., Volume 39, C96-C115.

(4) **Xiao, J., Gu, M., and Langou, J., 2017:** Fast parallel randomized QR with column pivoting algorithms for reliable low-rank matrix approximations. IEEE 24th International Conference on High Performance Computing (HiPC), IEEE, 2017, 233-242.

(5) **Feng, Y., Xiao, J., and Gu, M., 2019:** Flip-flop spectrum-revealing QR factorizations and its applications to singular value decomposition. Electronic Transactions on Numerical Analysis (ETNA), Volume 51, 469-494.

(6) **Duersch, J.A., and Gu, M., 2020:** Randomized projection for rank-revealing matrix factorizations and low-rank approximations. SIAM Review, Volume 62, Issue 3, 661-682.

(7) **Huckaby, D.A., and Chan, T.F., 2003:** On the convergence of Stewart's QLP algorithm for approximating the SVD. Numer. Algorithms, Volume 32, 287-316.

(8) **Huckaby, D.A., and Chan, T.F., 2005:** Stewart's pivoted QLP decomposition for low-rank matrices Numerical Linear Algebra with Applications, Volume 12, 153-159.

6.19.54 subroutine rqlp_svd_cmp2 (mat, s, leftvec, rightvec, failure, niter, rng_alg, nover, nover_svd, maxiter, max_francis_steps, perfect_shift, bisect, dqds)

Purpose

RQLP_SVD_CMP2 computes approximations of the nsvd largest singular values and associated left and right singular vectors of a full m-by-n real matrix MAT using a four-step procedure, which can be termed a randomized QLP-SVD algorithm:

- First, an approximate and randomized partial QLP factorization of MAT is computed as

  MAT = Q * L * P

  , where Q is a m-by-k matrix with orthonormal colmuns, P is an k-by-n matrix with orthonormal rows and L is a lower k-by-k triangular matrix.
- In a second step, the matrix product MAT * P' is computed and, at the user option, a number of QR-QL iterations are performed (this is equivalent to subspace iterations) to improve the estimates of the principal row and columns subspaces of MAT.
- In a third step, a Singular Value Decomposition (SVD) of the matrix product MAT * P' is computed. The singular values and left singular vectors of this SVD are also estimates of the singular values and left singular vectors of MAT.
- In a final step, estimates of the associated right singular vectors of MAT are obtained by pre-multiplying P' by the right singular vectors in the SVD of this matrix product.

A very fast randomized partial and truncated QLP factorization is used in the first phase of the QLP-SVD algorithm.

nsvd is the target rank of the partial SVD, which is sought, and is equal to the size of the output real vector argument S, i.e., nsvd = size(S).

See Further Details and the cited references for more information.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is not modified by the routine.

S (OUTPUT) real(stnd), dimension(:) On exit, S(:) contains the first top nsvd singular values of MAT. The singular values are given in decreasing order and are positive or zero.

The size of S must verify:

- size(S) = nsvd <= min(size(MAT,1) , size(MAT,2)) .

LEFTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed top nsvd left singular vectors of MAT. The left singular vector associated with the singular value S(j) is stored in the j-th column of LEFTVEC.

The shape of LEFTVEC must verify:

- size(LEFTVEC, 1) = size(MAT, 1) = m ,
- size(LEFTVEC, 2) = size(S) = nsvd .

RIGHTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed top nsvd right singular vectors of MAT. The right singular vector associated with the singular value S(j) is stored in the j-th column of RIGHTVEC.

The shape of RIGHTVEC must verify:

- size(RIGHTVEC, 1) = size(MAT, 2) = n ,
- size(RIGHTVEC, 2) = size(S) = nsvd .

FAILURE (OUTPUT, OPTIONAL) logical(lgl) On exit, if the optional logical argument FAILURE is present, a test of the accuracy of the computed singular triplets is performed and in that case:

- FAILURE = false : indicates successful exit;
- FAILURE = true : indicates that some singular values and vectors of MAT failed to converge in NITER QR-QL iterations.

If FAILURE = true on exit, results are still useful, but some of the approximated singular triplets may have a poor accuracy.

NITER (INPUT, OPTIONAL) integer(i4b) The number of subspace iterations performed in the subroutine after the initial QLP factorization and first subspace projection for computing the top nsvd singular triplets.

NITER must be positive or null.

By default, no subspace iterations are performed after the initial QLP factorization and first subspace projection.

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian matrix in the randomized (partial) QLP phase of the QLP-SVD algorithm if RANDOM_QR = true.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to RQLP_SVD_CMP.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

NOVER (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the randomized partial QLP phase of the QLP-SVD algorithm if RANDOM_QR = true.

NOVER must be positive or null and verify the relationship:

- NOVER + NOVER_SVD + size(S) <= size(MAT, 1).

and is adjusted if necessary to verify this relationship in all cases.

See Further Details and the cited references for the meaning and usefulness of the oversampling size in the partial randomized QR and QLP algorithms.

By default, the oversampling size is set to:

- max((NOVER_SVD+size(S))/2_i4b, 10).

NOVER_SVD (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the SVD phase of the QLP-SVD algorithm for computing the top nsvd singular triplets.

NOVER_SVD must be positive or null and verify the relationship:

- NOVER_SVD + size(S) <= min(size(MAT,1) , size(MAT,2))

and is adjusted if necessary to verify this relationship in all cases.

See Further Details and the cited references for the meaning and usefulness of the oversampling size in the SVD phase of the QLP-SVD algorithm.

By default, the oversampling size is set to 10.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm, which is used in the last step of the QLP-SVD algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) The optional argument MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g., QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm, which is used in the last step of the QLP-SVD algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

The default is the minimum of nsvd and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) The optional argument PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm, which is used in the last step of the QLP-SVD algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed QLP-SVD decomposition at the expense of a slightly slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

The QLP-SVD algorithm implemented in RQLP_SVD_CMP2 subroutine is a variation of the TXUV algorithm described in the references (2), (5) and (6) in which the initial randomized partial QR factorization is replaced by the randomized partial and truncated QR algorithm described in the reference (9).

With this modification, RQLP_SVD_CMP2 subroutine is less accurate than RQLP_SVD_CMP subroutine, but significantly faster and much less memory demanding.

For further details, on computing low-rank matrix approximations with a QLP factorization, the QLP-SVD (e.g., TXUV) algorithm or randomized (partial) QLP and QR factorizations, see:

(1) **Stewart, G.W., 1999:** The QLP approximation to the singular value decomposition. SIAM J. Sci. Comput., Volume 20, 1336-1348.

(2) **Duersch, J.A., and Gu, M., 2017:** Randomized QR with column pivoting. SIAM J. Sci. Comput., Volume 39, C263-C291.

(3) **Martinsson, P.G., Quintana-Orti, G., Heavner, N., and Van de Geijn, R., 2017:** Householder QR factorization with randomization for column pivoting (HQRRP). SIAM J. Sci. Comput., Volume 39, C96-C115.

(4) **Xiao, J., Gu, M., and Langou, J., 2017:** Fast parallel randomized QR with column pivoting algorithms for reliable low-rank matrix approximations. IEEE 24th International Conference on High Performance Computing (HiPC), IEEE, 2017, 233-242.

(5) **Feng, Y., Xiao, J., and Gu, M., 2019:** Flip-flop spectrum-revealing QR factorizations and its applications to singular value decomposition. Electronic Transactions on Numerical Analysis (ETNA), Volume 51, 469-494.

(6) **Duersch, J.A., and Gu, M., 2020:** Randomized projection for rank-revealing matrix factorizations and low-rank approximations. SIAM Review, Volume 62, Issue 3, 661-682.

(7) **Huckaby, D.A., and Chan, T.F., 2003:** On the convergence of Stewart's QLP algorithm for approximating the SVD. Numer. Algorithms, Volume 32, 287-316.

(8) **Huckaby, D.A., and Chan, T.F., 2005:** Stewart's pivoted QLP decomposition for low-rank matrices Numerical Linear Algebra with Applications, Volume 12, 153-159.

(9) **Mary, T., Yamazaki, I., Kurzak, J., Luszczek, P., Tomov, S., and Dongarra, J., 2015:** Performance of Random Sampling for Computing Low-rank Approximations of a Dense Matrix on GPUs. International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15).

6.19.55 subroutine rqlp_svd_cmp_fixed_precision (mat, relerr, s, leftvec, rightvec, failure, niter, random_qr, rng_alg, blk_size, nover, maxiter, max_francis_steps, perfect_shift, bisect, dqds)

Purpose

RQLP_SVD_CMP_FIXED_PRECISION computes approximations of the nsvd largest singular values and associated left and right singular vectors of a full m-by-n real matrix MAT using a four-step procedure, which can be termed a QLP-SVD algorithm:

- First, a partial QLP factorization of MAT is computed as

  MAT = Q * L * P

  , where Q is a m-by-k orthogonal matrix, P' is an n-by-k orthogonal matrix and L is a lower k-by-k triangular matrix.

- In a second step, the matrix product MAT * P' is computed and at, the user option, a number of QR-QL iterations are performed (this is equivalent to subspace iterations) to improve the estimates of the principal row and columns subspaces of MAT.

- In a third step, a Singular Value Decomposition (SVD) of the matrix product MAT * P' is computed. The singular values and left singular vectors of this SVD are also estimates of the top singular values and left singular vectors of MAT.

- In a final step, estimates of the associated right singular vectors of MAT are obtained by pre-multiplying P' by the right singular vectors in the SVD of this matrix product.

By default, a standard deterministic BLAS2 QR factorization with column pivoting is used in the first phase of the QLP step of the QLP-SVD algorithm. However, if the optional logical argument RANDOM_QR is used with the value true, a fast randomized partial QR factorization is used in the QLP step of the QLP-SVD algorithm.

nsvd is the target rank of the partial SVD, which is sought, and this partial SVD must have an approximation error which fulfills:

||MAT-rSVD||_F <= ||MAT||_F * relerr

, where rSVD is the computed partial SVD approximation, || ||_F is the Frobenius norm and relerr is a prescribed accuracy tolerance for the relative error of the computed partial SVD approximation, which is specified in the input real argument RELERR.

In other words, nsvd is not known in advance and is determined in the subroutine. This explains why the output real array arguments S, LEFTVEC and RIGHTVEC, which contain the computed singular values and associated singular vectors of MAT on exit, must be declared in the calling program as pointers.

On exit, nsvd is equal to the size of the output real pointer argument S, which contains the computed singular values, i.e., nsvd = size(S) and the relative error in the Frobenius norm of the computed partial SVD approximation is output in argument RELERR.

RQLP_SVD_CMP_FIXED_PRECISION first searches incrementally the best (e.g., smallest) partial QR approximation, which fulfills the prescribed accuracy tolerance for the relative error. More precisely, the rank of this partial QR approximation is increased progressively until the prescribed accuracy tolerance is satisfied. This partial QR approximation is then transformed in a partial QLP factorization, which is improved and adjusted precisely in a final step to obtain the smallest partial SVD approximation, which satisfies the prescribed tolerance.

In all cases the relative error of the computed partial SVD approximation is output in argument RELERR.

See Further Details and the cited references for more information.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is not modified by the routine.

RELERR (INPUT/OUTPUT) real(stnd) On entry, the requested accuracy tolerance for the relative error of the computed partial SVD approximation.

The preset value for RELERR must be greater than 4*epsilon(RELERR) and less than one.

On exit, RELERR contains the relative error of the computed partial SVD approximation in the Frobenius norm:

- RELERR = ||MAT-rSVD||_F / ||MAT||_F

S (OUTPUT) real(stnd), dimension(:), pointer On exit, S(:) contains estimates of the first top nsvd singular values of MAT. The singular values are given in decreasing order and are positive or zero.

The statut of the pointer S must not be undefined on entry. If, on entry, the pointer S is already allocated, it will be first deallocated and then reallocated with the correct size.

On exit, the size of the pointer S will verify:

- size(S) = nsvd <= min(size(MAT,1) , size(MAT,2)) .

LEFTVEC (OUTPUT) real(stnd), dimension(:,:), pointer On exit, the computed top nsvd left singular vectors of MAT. The left singular vector associated with the singular value S(j) is stored in the j-th column of LEFTVEC.

The statut of the pointer LEFTVEC must not be undefined on entry. If, on entry, the pointer LEFTVEC is already allocated, it will be first deallocated and then reallocated with the correct shape.

On exit, the shape of the pointer LEFTVEC will verify:

- size(LEFTVEC, 1) = size(MAT, 1) = m ,
- size(LEFTVEC, 2) = size(S) = nsvd .

RIGHTVEC (OUTPUT) real(stnd), dimension(:,:), pointer On exit, the computed top nsvd right singular vectors of MAT. The right singular vector associated with the singular value S(j) is stored in the j-th column of RIGHTVEC.

The statut of the pointer RIGHTVEC must not be undefined on entry. If, on entry, the pointer RIGHTVEC is already allocated, it will be first deallocated and then reallocated with the correct shape.

On exit, the shape of the pointer RIGHTVEC will verify:

- size(RIGHTVEC, 1) = size(MAT, 2) = n ,
- size(RIGHTVEC, 2) = size(S) = nsvd .

FAILURE (OUTPUT, OPTIONAL) logical(lgl) On exit, if the optional logical argument FAILURE is present, a test of the accuracy of the computed singular triplets is performed and in that case:

- FAILURE = false : indicates successful exit;
- FAILURE = true : indicates that some singular values and vectors of MAT failed to converge in NITER QR-QL iterations for the requested accuracy tolerance for the relative error of the computed partial SVD approximation.

If FAILURE = true on exit, results are still useful, but some of the approximated singular triplets have a poor accuracy.

NITER (INPUT, OPTIONAL) integer(i4b) The number of subspace iterations performed in the subroutine after the initial QLP factorization and first subspace projection for computing the top nsvd singular triplets.

NITER must be positive or null.

By default, no subspace iterations are performed after the initial QLP factorization and first subspace projection.

RANDOM_QR (INPUT, OPTIONAL) logical(lgl) On entry, if RANDOM_QR is used with the value true, a fast randomized partial QLP factorization is used in the first phase of the QLP-SVD algorithm.

By default, RANDOM_QR = false, i.e., a standard (partial) deterministic QLP factorization is used in the first phase of the QLP-SVD algorithm.

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian matrix in the randomized (partial) QLP phase of the QLP-SVD algorithm if RANDOM_QR = true.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to RQLP_SVD_CMP.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

BLK_SIZE (INPUT, OPTIONAL) integer(i4b) On entry, the block size used in the randomized partial QLP phase of the QLP-SVD algorithm if RANDOM_QR = true.

BLK_SIZE must be greater or equal to one and less than min(m,n) and must be set to a much smaller value than min(m,n) usually, depending also on the architecture of the computer.

See Further Details and the cited references for the meaning of the block size in the randomized (partial) QR or QLP algorithm.

By default, BLK_SIZE is set to min(BLKSZ_QR, min(m,n)), where parameter BLKSZ_QR is the default block size for QR related algorithms specified in module Select_Parameters.

NOVER (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the randomized partial QLP phase of the QLP-SVD algorithm if RANDOM_QR = true.

NOVER must be positive or null and verify the relationship:

- NOVER + BLK_SIZE <= size(MAT, 1)

and is adjusted if necessary to verify these relationships in all cases.

See Further Details and the cited references for the meaning and usefulness of the oversampling size in the partial randomized QR and QLP algorithms.

By default, the oversampling size is set to 10.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm, which is used in the last step of the QLP-SVD algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) The optional argument MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g., QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm, which is used in the last step of the QLP-SVD algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

The default is the minimum of nsvd and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) The optional argument PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm, which is used in the last step of the QLP-SVD algorithm.

See description of suboutine SVD_CMP for further details about this optional argument.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed QLP-SVD decomposition at the expense of a slightly slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

Usually, the problem of low-rank matrix approximation falls into two categories:

- the fixed-rank problem, where the rank parameter nsvd is given;
- the fixed-precision problem, where we seek a partial SVD factorization, rSVD, as small as possible such that

  ||MAT-rSVD||_F <= eps

  , where eps is a given accuracy tolerance.

RQLP_SVD_CMP_FIXED_PRECISION is dedicated to solve the fixed-precision problem. The fixed-rank problem can be solved by subroutines RQLP_SVD_CMP or RQLP_SVD_CMP2.

The QLP-SVD algorithm implemented in RQLP_SVD_CMP_FIXED_PRECISION subroutine is a variation of the TXUV algorithm described in the references (2), (5) and (6).

For further details, on computing low-rank matrix approximations with a QLP factorization, the QLP-SVD (e.g., TXUV) algorithm or randomized (partial) QLP and QR factorizations, see:

(1) **Stewart, G.W., 1999:** The QLP approximation to the singular value decomposition. SIAM J. Sci. Comput., Volume 20, 1336-1348.

(2) **Duersch, J.A., and Gu, M., 2017:** Randomized QR with column pivoting. SIAM J. Sci. Comput., Volume 39, C263-C291.

(3) **Martinsson, P.G., Quintana-Orti, G., Heavner, N., and Van de Geijn, R., 2017:** Householder QR factorization with randomization for column pivoting (HQRRP). SIAM J. Sci. Comput., Volume 39, C96-C115.

(4) **Xiao, J., Gu, M., and Langou, J., 2017:** Fast parallel randomized QR with column pivoting algorithms for reliable low-rank matrix approximations. IEEE 24th International Conference on High Performance Computing (HiPC), IEEE, 2017, 233-242.

(5) **Feng, Y., Xiao, J., and Gu, M., 2019:** Flip-flop spectrum-revealing QR factorizations and its applications to singular value decomposition. Electronic Transactions on Numerical Analysis (ETNA), Volume 51, 469-494.

(6) **Duersch, J.A., and Gu, M., 2020:** Randomized projection for rank-revealing matrix factorizations and low-rank approximations. SIAM Review, Volume 62, Issue 3, 661-682.

(7) **Huckaby, D.A., and Chan, T.F., 2003:** On the convergence of Stewart's QLP algorithm for approximating the SVD. Numer. Algorithms, Volume 32, 287-316.

(8) **Huckaby, D.A., and Chan, T.F., 2005:** Stewart's pivoted QLP decomposition for low-rank matrices Numerical Linear Algebra with Applications, Volume 12, 153-159.

6.19.56 subroutine qlp_cmp (mat, beta, tau, lmat, qmat, pmat, random_qr, truncated_qr, rng_alg, blk_size, nover)

Purpose

QLP_CMP computes a partial or complete QLP factorization of a m-by-n matrix MAT:

MAT = Q * L * P

, where Q is a m-by-krank orthogonal matrix, P is a krank-by-n orthogonal matrix and L is a krank-by-krank lower triangular matrix. If krank = min(m,n), the QLP factorization is complete and MAT = Q * L * P .

The QLP factorization is obtained by a two-step algorithm:

- first, a partial (or complete) QR factorization with column pivoting of MAT is computed;
- in a second step, a LQ Decomposition of the (permuted) upper triangular or trapezoidal (e.g., if n>m) factor, R, of this QR decomposition is computed.

By default, a standard deterministic QR factorization with column pivoting is used in the first phase of the QLP algorithm. However, if the optional logical argument RANDOM_QR is used with the value true, an alternate fast randomized (partial) QR factorization is used in the first phase of the QLP algorithm. Furthermore if, in addition, the optional logical argument TRUNCATED_QR is used with the value true, an even faster (but less accurate) randomized partial and truncated QR factorization will be used in the first phase of the QLP algorithm. In all cases, a deterministic blocked LQ factorization is used in the second step of the QLP factorization.

At the user option, the QLP factorization can also be only partial, e.g., the subroutine stops the computations when the numbers of columns of Q and of rows of P are equal to a predefined value equals to krank = size(BETA) = size(TAU).

The QLP decomposition provides a reasonable and cheap estimate of the Singular Value Decomposition (SVD) of a matrix when this matrix has a low rank or a significant gap in its singular values spectrum.

See Further Details and the cited references for more information.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real m-by-n matrix to be decomposed.

On exit, MAT has been overwritten by details of its (partial) QLP factorization.

See Further Details.

BETA (OUTPUT) real(stnd), dimension(:) On exit, the scalars factors of the elementary reflectors defining Q.

See Further Details.

The size of BETA must verify:

- size(BETA) = krank <= min(m , n) = min(size(MAT,1) , size(MAT,2)).

TAU (OUTPUT) real(stnd), dimension(:) On exit, the scalars factors of the elementary reflectors defining P.

See Further Details.

The size of TAU must verify:

- size(TAU) = krank <= min(m , n) = min(size(MAT,1) , size(MAT,2)).

LMAT (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, LMAT stores the lower triangular matrix L in the (partial) QLP factorization of MAT. The diagonal elements of LMAT (called the L-values) are estimates of the singular values of MAT if there is a significant gap in the singular values spectrum of MAT.

See Further Details.

The shape of LMAT must verify:

- size(LMAT, 1) = size(LMAT, 2) = krank.

QMAT (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, QMAT stores the first krank columns of the orthogonal matrix Q in the (partial) QLP factorization of MAT.

See Further Details.

The shape of QMAT must verify:

- size(QMAT, 1) = m.
- size(QMAT, 2) = krank.

PMAT (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, PMAT stores the first krank rows of the orthogonal matrix P in the (partial) QLP factorization of MAT.

See Further Details.

The shape of PMAT must verify:

- size(QMAT, 1) = krank.
- size(QMAT, 2) = n.

RANDOM_QR (INPUT, OPTIONAL) logical(lgl) On entry, if RANDOM_QR is used with the value true, a fast randomized (partial) QR factorization with column pivoting is used in the first phase of the QLP algorithm.

By default, RANDOM_QR = false, i.e., A standard deterministic (partial) QR factorization with column pivoting is used in the first phase of the QLP algorithm.

TRUNCATED_QR (INPUT, OPTIONAL) logical(lgl) On entry, if TRUNCATED_QR is used with the value true in addition to RANDOM_QR also set to true, a very fast (but less accurate) randomized partial and truncated QR factorization is used in the first phase of the QLP algorithm.

By default, TRUNCATED_QR = false, i.e., a "standard" randomized (partial) QR factorization with column pivoting is used in the first phase of the QLP algorithm if RANDOM_QR = true.

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian matrix in the randomized (partial) QR phase of the QLP algorithm, if RANDOM_QR = true.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to QLP_CMP.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

BLK_SIZE (INPUT, OPTIONAL) integer(i4b) On entry, the block size used in the randomized (partial) QR phase of the QLP algorithm, if RANDOM_QR = true (and TRUNCATED_QR = false).

BLK_SIZE must be greater or equal to one and less than min(m,n) and must be set to a much smaller value than min(m,n) usually, depending also on the architecture of the computer.

See Further Details and the cited references for the meaning of the block size in the randomized (partial) QR algorithm.

By default, BLK_SIZE is set to min(BLKSZ_QR, min(m,n)), where parameter BLKSZ_QR is the default block size for QR related algorithms specified in module Select_Parameters.

NOVER (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the randomized (partial) QR phase of the QLP algorithm, if RANDOM_QR = true.

NOVER must be positive or null and verify the relationships:

- NOVER + BLK_SIZE <= size(MAT, 1) if TRUNCATED_QR = false;
- NOVER + size(BETA) <= size(MAT, 1) if TRUNCATED_QR = true.

and is adjusted if necessary to verify these relationships in all cases.

See Further Details and the cited references for the meaning and usefulness of the oversampling size in the randomized (partial) QR algorithms.

By default, the oversampling size is set to:

- 10 if TRUNCATED_QR = false;
- max(size(BETA)/2_i4b, 10) if TRUNCATED_QR = true.

Further Details

QLP_CMP first computes a (partial or complete) QR factorization with column pivoting of the m-by-n matrix MAT:

MAT * N = Q * R

, where N is a n-by-n permutation matrix, R is a upper triangular or trapezoidal (i.e., if n>m) matrix and Q is a m-by-m orthogonal matrix.

If the optional logical argument RANDOM_QR is used with the value true, a fast randomized partial (and truncated, if the optional logical argument TRUNCATED_QR is also used with the value true) QR factorization with column pivoting is used in this first phase of the QLP algorithm.

At the user option, this QR factorization can also be only partial, e.g., the subroutine ends when the numbers of columns of Q is equal to a predefined value equals to krank = size(BETA) = size(TAU).

This leads implicitly to the following partition of Q:

[Q1 Q2]

where Q1 is a m-by-krank orthonormal matrix and Q2 is a m-by-(m-krank) orthonormal matrix orthogonal to Q1, and to the following corresponding partition of R:

[R11 R12]

[R21 R22]

where R11 is a krank-by-krank triangular matrix, R21 is zero by construction, R12 is a full krank-by-(n-krank) matrix and R22 is a full (m-krank)-by-(n-krank) matrix.

In a second step, QLP_CMP computes a deterministic LQ factorization of the matrix product:

R * N' = L * P

if the first QR factorization is complete, or of the matrix product:

[R11 R12] * N' = L * P

if this first QR factorization is only partial. This leads to the (partial) QLP factorization of MAT:

MAT = Q1 * L * P

where L is a krank-by-krank triangular matrix and P is a krank-by-n matrix with orthonormal rows.

The properties of the QLP factorization and when it can be used as a good proxy for the (partial or complete) SVD of a matrix are discussed in the references (2), (7), (8) and (9).

The computations are parallelized if OPENMP is used. However, note that QLP_CMP uses a standard "BLAS2" algorithm without any blocking for performing the first QR factorization with column pivoting if the optional logical argument RANDOM_QR is not used or used with the value false. On the other hand, QLP_CMP uses fast randomized and blocked QR algorithms with column pivoting (see the references (3), (4), (5) and (6)) if RANDOM_QR is used with the value true. These randomized algorithms are thus particularly efficient for large matrices.

The standard deterministic BLAS2 algorithm for computing a QR factorization with column pivoting is described in the reference (1). The randomized partial QR algorithm with column pivoting used if the optional logical argument RANDOM_QR is present with the value true is described in the references (3), (4) and (5). Finally, the randomized partial and truncated QR algorithm with column pivoting used if both the optional logical arguments RANDOM_QR and TRUNCATED_QR are present with the value true is described in the reference (6). This algorithm is the fastest, but less accurate than the randomized partial QR algorithm with column pivoting described in the references (3), (4) and (5).

In all cases, QLP_CMP uses an efficient (but deterministic) blocked algorithm for performing the LQ factorization in the second step of the QLP decomposition. The LQ factorization is described in the reference (1).

On exit, the matrix Q is represented as a product of elementary reflectors

Q = H(1) * H(2) * ... * H(krank), where krank = size(BETA) <= min(m , n).

Each H(i) has the form

H(i) = I + beta * (v * v') ,

where beta is a real scalar and v is a real m-element vector with v(1:i-1) = 0. v(i:m) is stored on exit in MAT(i:m,i) and beta in BETA(i). Note also that v(i) = 1.

On exit of QLP_CMP, the orthonormal matrix Q stored in factored form in MAT can be generated by a call to suboutine ORTHO_GEN_QR with arguments MAT and BETA. Alternatively, QLP_CMP computes the first krank columns of Q explicitly if the optional array argument QMAT is present.

The matrix P is represented as a product of elementary reflectors

Q = G(k) * ... * G(2) * G(1), where krank = size(TAU) <= min(m , n).

Each G(i) has the form

G(i) = I + tau * (u * u') ,

where tau is a real scalar and u is a real n-element vector with u(1:i-1) = 0. u(i:n) is stored on exit in MAT(i,i:n) and tau in TAU(i). Note also that u(i) = 1.

On exit of QLP_CMP, the orthonormal matrix P stored in factored in MAT can be generated by a call to suboutine ORTHO_GEN_LQ with arguments MAT and TAU. Alternatively, QLP_CMP computes the first krank rows of P explicitly if the optional array argument PMAT is present.

Finally, QLP_CMP outputs the krank-by-krank lower triangular matrix L in the optional array argument LMAT. If LMAT is not specified in the QLP_CMP call, the L factor of the QLP decomposition is not stored on exit.

For further details on the QLP factorization and its use, or randomized QR and QLP algorithms, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Stewart, G.W., 1999:** The QLP approximation to the singular value decomposition. SIAM J. Sci. Comput., Volume 20, 1336-1348.

(3) **Duersch, J.A., and Gu, M., 2017:** Randomized QR with column pivoting. SIAM J. Sci. Comput., Volume 39, C263-C291.

(4) **Martinsson, P.G., Quintana-Orti, G., Heavner, N., and Van de Geijn, R., 2017:** Householder QR factorization with randomization for column pivoting (HQRRP). SIAM J. Sci. Comput., Volume 39, C96-C115.

(5) **Duersch, J.A., and Gu, M., 2020:** Randomized projection for rank-revealing matrix factorizations and low-rank approximations. SIAM Review, Volume 62, Issue 3, 661-682.

(6) **Mary, T., Yamazaki, I., Kurzak, J., Luszczek, P., Tomov, S., and Dongarra, J., 2015:** Performance of Random Sampling for Computing Low-rank Approximations of a Dense Matrix on GPUs. International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15).

(7) **Wu, N., and Xiang, H., 2020:** Randomized QLP decomposition. Linear algebra and its applications, Volume 599, 18-35

(8) **Huckaby, D.A., and Chan, T.F., 2003:** On the convergence of Stewart's QLP algorithm for approximating the SVD. Numer. Algorithms, Volume 32, 287-316.

(9) **Huckaby, D.A., and Chan, T.F., 2005:** Stewart's pivoted QLP decomposition for low-rank matrices Numerical Linear Algebra with Applications, Volume 12, 153-159.

6.19.57 `subroutine qlp_cmp2 ( mat, lmat, qmat, pmat, niter_qrql, random_qr, truncated_qr, rng_alg, blk_size, nover )`

Purpose

QLP_CMP2 computes a partial or complete QLP factorization of an m-by-n matrix MAT:

MAT = Q * L * P

, where Q is a m-by-krank orthogonal matrix, P is a krank-by-n orthogonal matrix and L is a krank-by-krank lower triangular matrix. If krank = min(m,n), the QLP factorization is complete and MAT = Q * L * P .

The QLP factorization is obtained by a three-step algorithm:

- first, a partial (or complete) QR factorization with column pivoting of MAT is computed;
- in a second step, a LQ Decomposition of the (permuted) upper triangular or trapezoidal (e.g., if n>m) factor, R, in this QR decomposition of MAT is computed.
- and, in a final step, NITER_QRQL QR-QL iterations can be performed on the L factor in this LQ decomposition to improve the accuracy of the diagonal elements of L (the so called L-values) as estimates of the singular values of MAT (see references (2), (7), (8) and (9) for details).

By default, a standard deterministic QR factorization with column pivoting is used in the first phase of the QLP algorithm. However, if the optional logical argument RANDOM_QR is used with the value true, an alternate fast randomized (partial) QR factorization is used in the first phase of the QLP algorithm. Furthermore if, in addition, the optional logical argument TRUNCATED_QR is used with the value true, an even faster (but less accurate) randomized partial and truncated QR factorization will be used in the first phase of the QLP algorithm. In all cases, deterministic blocked LQ and QR factorizations are used in the second and third steps of the QLP factorization.

At the user option, the QLP factorization can also be only partial, e.g., the subroutine stops the computations when the numbers of columns of Q and of rows of P are equal to a predefined value equals to krank = size(LMAT, 1) = size(LMAT, 2).

The QLP decomposition provides a reasonable and cheap estimate of the Singular Value Decomposition (SVD) of a matrix when this matrix has a low rank or a significant gap in its singular values spectrum.

See Further Details and the cited references for more information.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real m-by-n matrix to be decomposed.

On exit, MAT is destroyed as MAT is used as workspace in the routine.

See Further Details.

LMAT (OUTPUT) real(stnd), dimension(:,:) On exit, LMAT stores the lower triangular matrix L in the (partial) QLP factorization of MAT.

See Further Details.

The shape of LMAT must verify:

- size(LMAT, 1) = size(LMAT, 2) = krank.

QMAT (OUTPUT) real(stnd), dimension(:,:) On exit, QMAT stores the first krank columns of the orthogonal matrix Q in the (partial) QLP factorization of MAT.

See Further Details.

The shape of QMAT must verify:

- size(QMAT, 1) = m.
- size(QMAT, 2) = krank.

PMAT (OUTPUT) real(stnd), dimension(:,:) On exit, PMAT stores the first krank rows of the orthogonal matrix P in the (partial) QLP factorization of MAT.

See Further Details.

The shape of PMAT must verify:

- size(QMAT, 1) = krank.
- size(QMAT, 2) = n.

NITER_QRQL (INPUT, OPTIONAL) integer(i4b) The number of QR-QL iterations performed on L after the initial QLP factorization for improving the accuracy of the L-values. NITER_QRQL must be positive or null.

By default, no QR-QL iterations are performed after the initial QLP factorization.

RANDOM_QR (INPUT, OPTIONAL) logical(lgl) On entry, if RANDOM_QR is used with the value true, a fast randomized (partial) QR factorization with column pivoting is used in the first phase of the QLP algorithm.

By default, RANDOM_QR = false, i.e., A standard deterministic (partial) QR factorization with column pivoting is used in the first phase of the QLP algorithm.

TRUNCATED_QR (INPUT, OPTIONAL) logical(lgl) On entry, if TRUNCATED_QR is used with the value true in addition to RANDOM_QR also set to true, a very fast (but less accurate) randomized partial and truncated QR factorization is used in the first phase of the QLP algorithm.

By default, TRUNCATED_QR = false, i.e., a "standard" randomized (partial) QR factorization with column pivoting is used in the first phase of the QLP algorithm if RANDOM_QR = true.

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian matrix in the randomized (partial) QR phase of the QLP algorithm, if RANDOM_QR = true.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to QLP_CMP2.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

BLK_SIZE (INPUT, OPTIONAL) integer(i4b) On entry, the block size used in the randomized (partial) QR phase of the QLP algorithm, if RANDOM_QR = true (and TRUNCATED_QR = false).

BLK_SIZE must be greater or equal to one and less than min(m,n) and must be set to a much smaller value than min(m,n) usually, depending also on the architecture of the computer.

See Further Details and the cited references for the meaning of the block size in the randomized (partial) QR algorithm.

By default, BLK_SIZE is set to min(BLKSZ_QR, min(m,n)), where parameter BLKSZ_QR is the default block size for QR related algorithms specified in module Select_Parameters.

NOVER (INPUT, OPTIONAL) integer(i4b) The oversampling size used in the randomized (partial) QR phase of the QLP algorithm, if RANDOM_QR = true.

NOVER must be positive or null and verify the relationships:

- NOVER + BLK_SIZE <= size(MAT, 1) if TRUNCATED_QR = false;

- NOVER + size(LMAT, 1) <= size(MAT, 1) if TRUNCATED_QR = true.

and is adjusted if necessary to verify these relationships in all cases.

See Further Details and the cited references for the meaning and usefulness of the oversampling size in the randomized partial QR algorithm.

By default, the oversampling size is set to:

- 10 if TRUNCATED_QR = false;
- max(size(LMAT,1)/2_i4b, 10) if TRUNCATED_QR = true.

Further Details

QLP_CMP2 first computes a (partial or complete) QR factorization with column pivoting of the m-by-n matrix MAT:

MAT * N = Q * R

, where N is a n-by-n permutation matrix, R is a upper triangular or trapezoidal (i.e., if n>m) matrix and Q is a m-by-m orthogonal matrix.

If the optional logical argument RANDOM_QR is used with the value true, a fast randomized partial (and truncated, if the optional logical argument TRUNCATED_QR is also used with the value true) QR factorization with column pivoting is used in this first phase of the QLP algorithm.

At the user option, this QR factorization can also be only partial, e.g., the subroutine ends when the numbers of columns of Q is equal to a predefined value equals to krank = size(LMAT, 1) = size(LMAT, 2).

This leads implicitly to the following partition of Q:

[Q1 Q2]

where Q1 is a m-by-krank orthonormal matrix and Q2 is a m-by-(m-krank) orthonormal matrix orthogonal to Q1, and to the following corresponding partition of R:

[R11 R12]

[R21 R22]

where R11 is a krank-by-krank triangular matrix, R21 is zero by construction, R12 is a full krank-by-(n-krank) matrix and R22 is a full (m-krank)-by-(n-krank) matrix.

In a second step, QLP_CMP2 computes a deterministic LQ factorization of the matrix product:

R * N' = L * P

if the first QR factorization is complete, or of the matrix product:

[R11 R12] * N' = L * P

if this first QR factorization is only partial. This leads to the (partial) QLP factorization of MAT:

MAT = Q1 * L * P

where L is a krank-by-krank triangular matrix and P is a krank-by-n matrix with orthonormal rows.

In a final step, NITER_QRQL QR-QL iterations can be performed on L to improve the accuracy of the diagonal elements of L (the so called L-values) as estimates of the singular values of MAT (see references (2), (7), (8) and (9) for details) and the orthogonal matrices Q and P are updated accordingly.

The properties of the QLP factorization and when it can be used as a good proxy for the (partial or complete) SVD of a matrix are discussed in the references (2), (7), (8) and (9).

The computations are parallelized if OPENMP is used. However, note that QLP_CMP2 uses a standard "BLAS2" algorithm without any blocking for performing the first QR factorization with column pivoting if the optional logical argument RANDOM_QR is not used or used with the value false. On the other hand, QLP_CMP2 uses an efficient randomized and blocked QR algorithm with column pivoting (see the references (2), (3) and (4)) if RANDOM_QR is used with the value true. This randomized algorithm is thus particularly efficient for large matrices.

The standard deterministic BLAS2 algorithm for computing a QR factorization with column pivoting is described in the reference (1). The randomized partial QR algorithm with column pivoting used if the optional logical argument RANDOM_QR is present with the value true is described in the references (3), (4) and (5). Finally, the randomized partial and truncated QR algorithm with column pivoting used if both the optional logical arguments RANDOM_QR and TRUNCATED_QR are present with the value true is described in the reference (6). This algorithm is the fastest, but less accurate than the randomized partial QR algorithm with column pivoting described in the references (3), (4) and (5).

In all cases, QLP_CMP2 uses efficient blocked algorithms for performing the LQ factorization in the second step of the QLP algorithm and also in the final QR-QL iterations performed on L. The LQ factorization is described in the reference (1).

On exit, QLP_CMP2 stores:

- the krank-by-krank lower triangular matrix L in the array argument LMAT;
- the first krank columns of Q in the array argument QMAT;
- and the first krank rows of P in the array argument PMAT.

For further details on the QLP factorization and its use, randomized QR and QLP algorithms or QR-QL iterations, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(2) **Stewart, G.W., 1999:** The QLP approximation to the singular value decomposition. SIAM J. Sci. Comput., Volume 20, 1336-1348.

(3) **Duersch, J.A., and Gu, M., 2017:** Randomized QR with column pivoting. SIAM J. Sci. Comput., Volume 39, C263-C291.

(4) **Martinsson, P.G., Quintana-Orti, G., Heavner, N., and Van de Geijn, R., 2017:** Householder QR factorization with randomization for column pivoting (HQRRP). SIAM J. Sci. Comput., Volume 39, C96-C115.

(5) **Duersch, J.A., and Gu, M., 2020:** Randomized projection for rank-revealing matrix factorizations and low-rank approximations. SIAM Review, Volume 62, Issue 3, 661-682.

(6) **Mary, T., Yamazaki, I., Kurzak, J., Luszczek, P., Tomov, S., and Dongarra, J., 2015:** Performance of Random Sampling for Computing Low-rank Approximations of a Dense Matrix on GPUs. International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15).

(7) **Wu, N., and Xiang, H., 2020:** Randomized QLP decomposition. Linear algebra and its applications, Volume 599, 18-35

(8) **Huckaby, D.A., and Chan, T.F., 2003:** On the convergence of Stewart's QLP algorithm for approximating the SVD. Numer. Algorithms, Volume 32, 287-316.

(9) **Huckaby, D.A., and Chan, T.F., 2005:** Stewart's pivoted QLP decomposition for low-rank matrices Numerical Linear Algebra with Applications, Volume 12, 153-159.

6.19.58 subroutine rqlp_cmp (mat, lmat, qmat, pmat, niter, rng_alg, ortho, niter_qrql)

Purpose

RQLP_CMP computes a randomized partial QLP factorization of an m-by-n matrix MAT:

MAT = Q * L * P

, where Q is a m-by-krank matrix with orthonormal columns, P is a krank-by-n matrix with orthonormal rows and L is a krank-by-krank lower triangular matrix.

The randomized QLP factorization is only partial, e.g., the subroutine stops the computations when the numbers of columns of Q and of rows of P are equal to a predefined value equals to krank = size(LMAT, 1) = size(LMAT, 2).

The randomized partial QLP factorization is obtained by a four-step algorithm:

- first, the routines computes a partial QB factorization of MAT with the help of a randomized algorithm:

  MAT = Q * B

  , where Q is a m-by-krank orthonormal matrix, B is a krank-by-n matrix and the product Q*B is a good approximation of MAT according to the spectral or Frobenius norm;
- second, a QR factorization with column pivoting of B is computed and Q is post-multiplied by the krank-by-krank orthogonal matrix, O, in this QR factorization of B;
- in a third step, a LQ Decomposition of the (permuted) upper trapezoidal factor, R, in this QR decomposition of B is computed.
- and, in a final step, NITER_QRQL QR-QL iterations are performed on the L matrix in this LQ decomposition to improve the accuracy of the diagonal elements of L (the so called L-values) as estimates of the singular values of MAT (see references (5), (6) and (7) for details).

This randomized QLP decomposition provides a reasonable and cheap estimate of the Singular Value Decomposition (SVD) of a matrix when this matrix has a low rank or a significant gap in its singular values spectrum.

See Further Details and the cited references for more information.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the real m-by-n matrix to be decomposed.

MAT is not modified by the routine.

LMAT (OUTPUT) real(stnd), dimension(:,:) On exit, LMAT stores the lower triangular matrix L in the (partial) QLP factorization of MAT.

See Further Details.

The shape of LMAT must verify:

- size(LMAT, 1) = size(LMAT, 2) = krank.

QMAT (OUTPUT) real(stnd), dimension(:,:) On exit, QMAT stores the first krank columns of the orthogonal matrix Q in the (partial) QLP factorization of MAT.

See Further Details.

The shape of QMAT must verify:

- size(QMAT, 1) = m.
- size(QMAT, 2) = krank.

PMAT (OUTPUT) real(stnd), dimension(:,:) On exit, PMAT stores the first krank rows of the orthogonal matrix P in the (partial) QLP factorization of MAT.

See Further Details.

The shape of PMAT must verify:

- size(QMAT, 1) = krank.
- size(QMAT, 2) = n.

NITER (INPUT, OPTIONAL) integer(i4b) The number of randomized power or subspace iterations performed in the first phase of the randomized QLP algorithm for computing the preliminary randomized QB factorization.

NITER must be positive or null.

By default, 5 randomized power or subspace iterations are performed.

RNG_ALG (INPUT, OPTIONAL) integer(i4b) On entry, a scalar integer to select the random (uniform) number generator used to build the random gaussian test matrix in the initial randomized partial QB factorization.

The possible values are:

- ALG=1 : selects the Marsaglia's KISS random number generator;
- ALG=2 : selects the fast Marsaglia's KISS random number generator;
- ALG=3 : selects the L'Ecuyer's LFSR113 random number generator;
- ALG=4 : selects the Mersenne Twister random number generator;
- ALG=5 : selects the maximally equidistributed Mersenne Twister random number generator;
- ALG=6 : selects the extended precision of the Marsaglia's KISS random number generator;
- ALG=7 : selects the extended precision of the fast Marsaglia's KISS random number generator;
- ALG=8 : selects the extended precision of the L'Ecuyer's LFSR113 random number generator.
- ALG=9 : selects the extended precision of Mersenne Twister random number generator;
- ALG=10 : selects the extended precision of maximally equidistributed Mersenne Twister random number generator;

For other values, the current random number generator and its current state are not changed. Note further, that, on exit, the current random number generator is not reset to its previous value before the call to RQLP_CMP.

See the documentation of subroutine RANDOM_SEED_ in module Random for further information.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, orthonormalization is carried out between each step of the power iterations, to avoid loss of accuracy due to rounding errors. This means that subspace iterations are used instead of power iterations in the QB phase of the algorithm,
- ORTHO=false, orthonormalization is not performed.

The default is to use orthonormalization, e.g., ORTHO=true.

NITER_QRQL (INPUT, OPTIONAL) integer(i4b) The number of QR-QL iterations performed on L after the initial QLP factorization for improving the accuracy of the L-values. NITER_QRQL must be positive or null.

By default, no QR-QL iterations are performed after the initial QLP factorization.

Further Details

RQLP_CMP first computes a partial QB factorization of MAT with the help of a randomized algorithm:

MAT = Q * B

, where Q is a m-by-krank orthonormal matrix, B is a krank-by-n matrix and the product Q*B is a good approximation of MAT according to the spectral or Frobenius norm. Here, krank = size(LMAT, 1) = size(LMAT, 2).

In a second step, RQLP_CMP computes a deterministic QR factorization with column pivoting of the krank-by-n matrix B, to obtain an approximate QR factorization with column pivoting of MAT :

MAT * N = Q * (B * N) = Q * (O * R) = (Q * O) * R

, where N is a n-by-n permutation matrix, O is a krank-by-krank orthogonal matrix and R is a krank-by-n upper trapezoidal matrix.

In a third step, RQLP_CMP computes a deterministic LQ factorization of the matrix product:

R * N' = L * P

This leads to the approximate QLP factorization of MAT:

MAT = (Q * O) * L * P

where Q is a m-by-krank matrix with orthonormal columns, O is a krank-by-krank orthogonal matrix, L is a krank-by-krank lower triangular matrix and P is a krank-by-n matrix with orthonormal rows.

In a final step, NITER_QRQL QR-QL iterations can be performed on L to improve the accuracy of the diagonal elements of L (the so called L-values) as estimates of the singular values of MAT (see references (1), (5), (6) and (7) for details) and the orthogonal matrices Q and P are updated accordingly.

The computations are parallelized if OPENMP is used.

In all cases, RQLP_CMP uses efficient blocked algorithms for performing the QB, QR and LQ steps in the randomized QLP algorithm and also in the final QR-QL iterations performed on L.

On exit, RQLP_CMP stores:

- the krank-by-krank lower triangular matrix L in the array argument LMAT;
- the first krank columns of Q in the array argument QMAT;
- and the first krank rows of P in the array argument PMAT.

For further details on the QLP factorization and its use or randomized QB, QR and QLP algorithms or QR-QL iterations, see:

(1) **Stewart, G.W., 1999:** The QLP approximation to the singular value decomposition. SIAM J. Sci. Comput., Volume 20, 1336-1348.

(2) **Duersch, J.A., and Gu, M., 2017:** Randomized QR with column pivoting. SIAM J. Sci. Comput., Volume 39, C263-C291.

(3) **Martinsson, P.G., Quintana-Orti, G., Heavner, N., and Van de Geijn, R., 2017:** Householder QR factorization with randomization for column pivoting (HQRRP). SIAM J. Sci. Comput., Volume 39, C96-C115.

(4) **Martinsson, P.G., and Voronin, S., 2016:** A randomized blocked algorithm for efficiently computing rank-revealing factorizations of matrices. SIAM J. Sci. Comput., 38:5, S485-S507.

(5) **Wu, N., and Xiang, H., 2020:** Randomized QLP decomposition. Linear algebra and its applications, Volume 599, 18-35

(6) **Huckaby, D.A., and Chan, T.F., 2003::** On the convergence of Stewart's QLP algorithm for approximating the SVD. Numer. Algorithms, Volume 32, 287-316.

(7) **Huckaby, D.A., and Chan, T.F., 2005:** Stewart's pivoted QLP decomposition for low-rank matrices Numerical Linear Algebra with Applications, Volume 12, 153-159.

6.19.59 function maxdiag_gkinv_qr (e, lambda)

Purpose

This function computes the index of the element of maximum absolute value in the diagonal entries of

(GK - LAMBDA * I)**(-1)

where GK is a n-by-n symmetric tridiagonal matrix with a zero diagonal, I is the identity matrix and LAMBDA is a scalar.

Arguments

E (INPUT) real(stnd), dimension(:) On entry, the n-1 off-diagonal elements of the tridiagonal matrix.

LAMBDA (INPUT) real(stnd) On entry, the eigenvalue or shift used in the QR factorization.

Further Details

The diagonal entries of (GK - LAMBDA * I)**(-1) are computed by means of the QR factorization of (GK - LAMBDA * I). For the latter computation, the semiseparable structure of (GK - LAMBDA * I)**(-1) is used, see the reference (1). Moreover, it is assumed that GK is unreduced, but no check is done in the subroutine to verify this assumption.

This subroutine is adapted from the pseudo-code trace_Tinv given in the reference (1).

For further details, see:

(1) **Bini, D.A., Gemignani, L., and Tisseur, F., 2005:** The Ehrlich-Aberth method for the nonsymmetric tridiagonal eigenvalue problem. SIAM J. Matrix Anal. Appl., 27, 153-175.

(2) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(3) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

6.19.60 function maxdiag_gkinv_ldu (e, lambda)

Purpose

This function computes the index of the element of maximum absolute value in the diagonal entries of

(GK - LAMBDA * I)**(-1)

where GK is a n-by-n symmetric tridiagonal matrix with a zero diagonal, I is the identity matrix and LAMBDA is a scalar.

Arguments

E (INPUT) real(stnd), dimension(:) On entry, the n-1 off-diagonal elements of the tridiagonal matrix.

LAMBDA (INPUT) real(stnd) On entry, the eigenvalue or shift used.

Further Details

The diagonal entries of (GK - LAMBDA * I)**(-1) are computed by means of two triangular factorizations of (GK - LAMBDA * I) of the forms L(+) * D(+) * U(+) and U(-) * D(-) * L(-) where L(+) and L(-) are unit lower bidiagonal, U(+) and U(-) are unit upper bidiagonal, and D(+) and D(-) are diagonal.

It is assumed that GK is unreduced, but no check is done in the subroutine to verify this assumption.

This subroutine is adapted from the references (1) and (2).

For further details, on Fernando's method for computing eigenvectors of tridiagonal matrices, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(2) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

6.19.61 subroutine gk_qr_cmp (e, lambda, cs, sn, diag, sup1, sup2, maxdiag_gkinv)

Purpose

GK_QR_CMP factorizes the symmetric matrix GK - LAMBDA * I, where GK is an n-by-n symmetric tridiagonal matrix with a zero diagonal, I is the identity matrix and LAMBDA is a scalar, as

GK - LAMBDA * I = Q * R

where Q is an orthogonal matrix represented as the product of n-1 Givens rotations and R is an upper triangular matrix with at most two non-zero super-diagonal elements per column.

The parameter LAMBDA is included in the routine so that GK_QR_CMP may be used to obtain eigenvectors of GK by inverse iteration.

The subroutine also computes the index of the entry of maximum absolute value in the diagonal of (GK - LAMBDA * I)**(-1), which provides a good initial approximation to start the inverse iteration process for computing the eigenvector associated with the eigenvalue LAMBDA, see the references (1), (2) and (3) for further details.

Arguments

E (INPUT) real(stnd), dimension(:) On entry, the n-1 off-diagonal elements of the tridiagonal matrix.

LAMBDA (INPUT) real(stnd) On entry, the eigenvalue or shift used in the QR factorization.

CS (OUTPUT) real(stnd), dimension(:) On exit, the vector of the cosines coefficients of the chain of n-1 Givens rotations for the QR factorization of GK - LAMBDA * I.

The size of CS must be size(CS) = size(E) = n - 1.

SN (OUTPUT) real(stnd), dimension(:) On exit, the vector of the sines coefficients of the chain of n-1 Givens rotations for the QR factorization of GK - LAMBDA * I.

The size of SN must be size(SN) = size(E) = n - 1.

DIAG (OUTPUT) real(stnd), dimension(:) On exit, DIAG(:) contains the n diagonal elements of the upper triangular matrix R of the QR factorization of GK - LAMBDA * I.

The size of DIAG must verify: size(DIAG) = size(E) + 1 = n .

SUP1 (OUTPUT) real(stnd), dimension(:) On exit, SUP1(:n-1) contains the n-1 superdiagonal elements of the upper triangular matrix R of the QR factorization of GK - LAMBDA * I, SUP1(n) is arbitrary .

The size of SUP1 must verify: size(SUP1) = size(E) + 1 = n .

SUP2 (OUTPUT) real(stnd), dimension(:) On exit, SUP2(:n-2) contains the n-2 second superdiagonal elements of the upper triangular matrix R of the QR factorization of GK - LAMBDA * I, SUP2(n-1:n) is arbitrary .

The size of SUP2 must verify: size(SUP2) = size(E) + 1 = n .

MAXDIAG_GKINV (OUPTPUT) integer(i4b) On exit, MAXDIAG_GKINV is the index of the entry of maximum modulus in the main diagonal of (GK - LAMBDA * I)**(-1).

Further Details

The QR factorization of (GK - LAMBDA * I) is obtained by means of n-1 unitary Givens rotations.

The diagonal entries of (GK - LAMBDA * I)**(-1) are computed by means of this QR factorization of (GK - LAMBDA * I). For the latter computation, the semiseparable structure of (GK - LAMBDA * I)**(-1) is used, see the reference (1). Moreover, it is assumed that GK is unreduced for computing the index of the entry of maximum absolute value in the diagonal of (GK - LAMBDA * I)**(-1), but no check is done in the subroutine to verify this assumption.

For further details, see:

(1) **Bini, D.A., Gemignani, L., and Tisseur, F., 2005:** The Ehrlich-Aberth method for the nonsymmetric tridiagonal eigenvalue problem. SIAM J. Matrix Anal. Appl., 27, 153-175.

(2) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(3) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

6.19.62 subroutine bd_inviter (upper, d, e, s, leftvec, rightvec, failure, maxiter, scaling, initvec)

Purpose

BD_INVITER computes the left and right singular vectors of a real n-by-n bidiagonal matrix BD corresponding to a specified singular value, using Fernando's method and inverse iteration on the tridiagonal Golub-Kahan (TGK) form of the bidiagonal matrix BD.

Arguments

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : BD is upper bidiagonal ;
- UPPER = false : BD is lower bidiagonal.

D (INPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix BD.

E (INPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the bidiagonal matrix BD. E(1) is arbitrary.

The size of E must verify: size(E) = size(D) = n .

S (INPUT) real(stnd) On entry, the selected singular value of the bidiagonal matrix BD. The singular value must be positive or zero.

LEFTVEC (OUTPUT) real(stnd), dimension(:) On exit, the computed left singular vector.

The shape of LEFTVEC must verify: size(LEFTVEC) = size(D) = n .

RIGHTVEC (OUTPUT) real(stnd), dimension(:) On exit, the computed right singular vector.

The shape of RIGHTVEC must verify: size(RIGHTVEC) = size(D) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = FALSE : indicates successful exit,
- FAILURE = TRUE : indicates that some singular vectors failed to converge in MAXITER iterations.

MAXITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed in the subroutine. By default, 2 inverse iterations are performed.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if:

- SCALING=true, the bidiagonal matrix BD is scaled before computing the singular vector;
- SCALING=false, the bidiagonal matrix BD is not scaled.

The default is to scale the bidiagonal matrix.

INITVEC (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITVEC=true, a Fernando vector is used to start the inverse iteration process for computing the singular vectors of the bidiagonal matrix BD (e.g. the eigenvector of the associated tridiagonal Golub-Kahan matrix);
- INITVEC=false, a random uniform starting vector is used.

The default is to use a Fernando starting vector if the Golub-Kahan form of the input bidiagonal matrix is unreduced, and a random uniform starting vector otherwise.

Further Details

A first estimate of the singular vectors is computed by the Fernando method applied to the tridiagonal Golub-Kahan matrix associated with the bidiagonal matrix BD (see the reference (1) for details) if this Golub-Kahan form of the input bidiagonal matrix is unreduced. Otherwise, a random start is used as a first estimate of the singular vectors as in the standard inverse-iteration algorithm.

The singular vectors are then computed or refined using inverse iteration on the tridiagonal Golub-Kahan matrix.

For further details, on Fernando's method for computing eigenvectors of tridiagonal matrices or inverse iteration, see

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(2) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

(3) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(4) **Bini, D.A., Gemignani, L., and Tisseur, F., 2005:** The Ehrlich-Aberth method for the nonsymmetric tridiagonal eigenvalue problem. SIAM J. Matrix Anal. Appl., 27, 153-175.

6.19.63 subroutine bd_inviter (upper, d, e, s, leftvec, rightvec, failure, maxiter, ortho, backward_sweep, scaling, initvec)

Purpose

BD_INVITER computes the left and right singular vectors of a real n-by-n bidiagonal matrix BD corresponding to specified singular values, using Fernando's method and inverse iteration on the tridiagonal Golub-Kahan (TGK) form of the bidiagonal matrix BD.

Arguments

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : BD is upper bidiagonal ;
- UPPER = false : BD is lower bidiagonal.

D (INPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix BD.

E (INPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the bidiagonal matrix BD. E(1) is arbitrary.

The size of E must verify: size(E) = size(D) = n .

S (INPUT) real(stnd), dimension(:) On entry, selected singular values of the bidiagonal matrix BD. The singular values must be given in decreasing order and must be positive or zero.

The size of S must verify: size(S) <= size(D) = n .

LEFTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed left singular vectors. The left singular vector associated with the singular value S(j) is stored in the j-th column of LEFTVEC.

The shape of LEFTVEC must verify:

- size(LEFTVEC, 1) = size(D) = n ,
- size(LEFTVEC, 2) = size(S) .

RIGHTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed right singular vectors. The right singular vector associated with the singular value S(j) is stored in the j-th column of RIGHTVEC.

The shape of RIGHTVEC must verify:

- size(RIGHTVEC, 1) = size(D) = n ,
- size(RIGHTVEC, 2) = size(S) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = FALSE : indicates successful exit,
- FAILURE = TRUE : indicates that some singular vectors failed to converge in MAXITER iterations.

MAXITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed in the subroutine. By default, 2 inverse iterations are performed for all the singular vectors.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, all the singular vectors are orthogonalized by the Modified Gram-Schmidt or QR algorithm;
- ORTHO=false, the singular vectors are not orthogonalized by the Modified Gram-Schmidt or QR algorithm.

The default is to orthogonalize the singular vectors only for the singular values, which are not well-separated.

BACKWARD_SWEEP (INPUT, OPTIONAL) logical(lgl) On entry, if:

- BACKWARD_SWEEP=true and the singular vectors are orthogonalized by the modified Gram-Schmidt algorithm, a backward sweep of the modified Gram-Schmidt algorithm is also performed;
- BACKWARD_SWEEP=false, a backward sweep is not performed.

The default is not to perform a backward sweep of the modified Gram-Schmidt algorithm.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if:

- SCALING=true, the bidiagonal matrix BD is scaled before computing the singular vectors;
- SCALING=false, the bidiagonal matrix BD is not scaled.

The default is to scale the bidiagonal matrix.

INITVEC (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITVEC=true, Fernando vectors are used to start the inverse iteration process for computing the singular vectors of the bidiagonal matrix BD (e.g. the eigenvectors of the associated Golub-Kahan tridiagonal matrix);
- INITVEC=false, random uniform starting vectors are used.

The default is to use Fernando starting vectors if the singular values are well-separated and the Golub-Kahan form of the input bidiagonal matrix is unreduced, and random uniform starting vectors otherwise.

Further Details

A first estimate of the singular vectors is computed by the Fernando method applied to the tridiagonal Golub-Kahan matrix associated with the bidiagonal matrix BD (see the reference (1) for details) for the singular values which are well-separated and if the Golub-Kahan form of the input bidiagonal matrix is unreduced. For the other singular values, a random start is used as a first estimate of the singular vectors as in the standard inverse-iteration algorithm.

The singular vectors are then computed or refined using inverse iteration on the tridiagonal Golub-Kahan matrix for all the singular values at one step.

By default, the singular vectors are then orthogonalized by the Modified Gram-Schmidt or QR algorithm only if the singular values are not well-separated.

The computation of the singular vectors is parallelized if OPENMP is used.

BD_INVITER may fail if clusters of tiny singular values are present in parameter S.

For further details, on Fernando's method for computing eigenvectors of tridiagonal matrices or inverse iteration, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(2) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

(3) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

(4) **Bini, D.A., Gemignani, L., and Tisseur, F., 2005:** The Ehrlich-Aberth method for the nonsymmetric tridiagonal eigenvalue problem. SIAM J. Matrix Anal. Appl., 27, 153-175.

6.19.64 subroutine bd_inviter2 (mat, tauq, taup, d, e, s, leftvec, rightvec, failure, maxiter, ortho, backward_sweep, scaling, initvec)

Purpose

BD_INVITER2 computes the left and right singular vectors of a full real m-by-n matrix MAT corresponding to specified singular values, using inverse iteration.

It is required that the original matrix MAT has been reduced to upper or lower bidiagonal form BD by an orthogonal transformation:

Q' * MAT * P = BD

where Q and P are orthogonal. This can be done with a call to BD_CMP with parameters TAUQ and TAUP, before calling BD_SVD, BD_SINGVAL or BD_SINGVAL2 subroutines for computing singular values and, finally, BD_INVITER2 for computing selected singular vectors.

Alternatively and more simply, the user can call SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 subroutines, with parameters TAUQ and TAUP present, to compute the bidiagonal reduction and singular values in one step and, finally, call BD_INVITER2 for computing all or selected singular vectors of MAT.

If m >= n, BD is upper bidiagonal and if m < n, BD is lower bidiagonal.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the original m-by-n matrix after reduction by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 subroutines. MAT must contains the vectors which define the elementary reflectors H(i) and G(i) whose products determine the matrices Q and P, as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 subroutines when the arguments TAUQ and TAUP are present in the call

to these subroutines. MAT must be specified as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 and is not modified by the routine.

TAUQ (INPUT) real(stnd), dimension(:) TAUQ(i) must contain the scalar factor of the elementary reflector H(i) which determines Q, as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 in their array argument TAUQ.

The size of TAUQ must verify: size(TAUQ) = min(size(MAT,1) , size(MAT,2)) .

TAUP (INPUT) real(stnd), dimension(:) TAUP(i) must contain the scalar factor of the elementary reflector G(i), which determines P, as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 in their array argument TAUP.

The size of TAUP must verify: size(TAUP) = min(size(MAT,1) , size(MAT,2)) .

D (INPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix BD as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2.

The size of D must verify: size(D) = min(size(MAT,1) , size(MAT,2)) .

E (INPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the bidiagonal matrix BD as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2:

- if m >= n, E(i) = BD(i-1,i) for i = 2,3,. . . ,n;
- if m < n, E(i) = BD(i,i-1) for i = 2,3,. . . ,m.

E(1) is arbitrary.

The size of E must verify: size(E) = min(size(MAT,1) , size(MAT,2)) .

S (INPUT) real(stnd), dimension(:) On entry, selected singular values of the bidiagonal matrix BD. The singular values must be given in decreasing order and are assumed to be positive or zero.

The size of S must verify: size(S) <= min(size(MAT,1) , size(MAT,2)) .

LEFTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed left singular vectors. The left singular vector associated with the singular value S(j) is stored in the j-th column of LEFTVEC.

The shape of LEFTVEC must verify:

- size(LEFTVEC, 1) = size(MAT, 1) = m ,
- size(LEFTVEC, 2) = size(S) .

RIGHTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed right singular vectors. The right singular vector associated with the singular value S(j) is stored in the j-th column of RIGHTVEC.

The shape of RIGHTVEC must verify:

- size(RIGHTVEC, 1) = size(MAT, 2) = n ,
- size(RIGHTVEC, 2) = size(S) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = FALSE : indicates successful exit,
- FAILURE = TRUE : indicates that some singular vectors of BD failed to converge in MAXITER iterations.

MAXITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed in the subroutine.

By default, 2 inverse iterations are performed for all the singular vectors.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, all the singular vectors of the bidiagonal matrix BD are orthogonalized by the Modified Gram-Schmidt or QR algorithm;
- ORTHO=false, the singular vectors of the bidiagonal matrix BD are not orthogonalized by the Modified Gram-Schmidt or QR algorithm.

The default is to orthogonalize the singular vectors only for the singular values, which are not well-separated.

BACKWARD_SWEEP (INPUT, OPTIONAL) logical(lgl) On entry, if:

- BACKWARD_SWEEP=true and the singular vectors of the bidiagonal matrix BD are orthogonalized by the modified Gram-Schmidt algorithm, a backward sweep of the modified Gram-Schmidt algorithm is also performed;
- BACKWARD_SWEEP=false, a backward sweep is not performed.

The default is not to perform a backward sweep of the modified Gram-Schmidt algorithm.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if:

- SCALING=true, the bidiagonal matrix BD is scaled before computing the singular vectors;
- SCALING=false, the bidiagonal matrix BD is not scaled.

The default is to scale the bidiagonal matrix.

INITVEC (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITVEC=true, Fernando vectors are used to start the inverse iteration process for computing the singular vectors of the bidiagonal matrix BD (e.g. the eigenvectors of the associated Golub-Kahan tridiagonal matrix);
- INITVEC=false, random uniform starting vectors are used.

The default is to use Fernando starting vectors if the singular values are well-separated and the Golub-Kahan form of the input bidiagonal matrix is unreduced, and random uniform starting vectors otherwise.

Further Details

A first estimate of the singular vectors is computed by the Fernando method applied to the tridiagonal Golub-Kahan matrix associated with the bidiagonal matrix BD (see the reference (1) for details) for the singular values which are well-separated and if the Golub-Kahan form of the input bidiagonal matrix is unreduced. For the other singular values, a random start is used as a first estimate of the singular vectors as in the standard inverse-iteration algorithm.

The singular vectors of BD are then computed or refined using inverse iteration on the tridiagonal Golub-Kahan matrix for all the singular values at one step.

By default, the singular vectors of BD are then orthogonalized by the Modified Gram-Schmidt or QR algorithm only if the singular values are not well-separated.

The singular vectors of MAT are finally computed by a blocked back-transformation algorithm.

The computation of the singular vectors of BD and the blocked back-transformation algorithm to find the singular vectors of MAT are parallelized if OPENMP is used.

BD_INVITER2 may fail if some singular values specified in parameter S are nearly identical for some pathological matrices.

For further details on Fernando method for computing eigenvectors of tridiagonal matrices, the blocked back-transformation algorithm or inverse iteration, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(2) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

(3) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.19.65 subroutine bd_inviter2 (mat, p, d, e, s, leftvec, rightvec, failure, maxiter, ortho, backward_sweep, scaling, initvec, tol_reortho)

Purpose

BD_INVITER2 computes the left and right singular vectors of a full real m-by-n matrix MAT with m>=n corresponding to specified singular values, using inverse iteration.

It is required that the original matrix MAT has been reduced to upper bidiagonal form BD by an orthogonal transformation:

Q' * MAT * P = BD

where Q and P are orthogonal. This can be done with a call to BD_CMP2 (or a call to BD_CMP followed by a call to ORTHO_GEN_BD), before calling BD_SVD, BD_SINGVAL or BD_SINGVAL2 subroutines for computing singular values and BD_INVITER2 for computing selected singular vectors.

Alternatively and more simply, the user can call SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 subroutines, with parameter P present, to compute the bidiagonal reduction and singular values in one step and, finally, call BD_INVITER2 for computing all or selected singular vectors of MAT.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the m-by-n orthogonal matrix Q after reduction by BD_CMP2, SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 (or by BD_CMP and ORTHO_GEN_BD). MAT is not modified by the routine.

The shape of MAT must verify: size(MAT, 1) >= size(MAT, 2) = n .

P (INPUT) real(stnd), dimension(:,:) On entry, the n-by-n orthogonal matrix P after reduction by BD_CMP2, SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 (or by BD_CMP and ORTHO_GEN_BD). If P has been computed by BD_CMP2, SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4, P can be stored in factored form or not. Both cases are handled by the subroutine. P is not modified by the routine.

The shape of P must verify: size(P, 1) = size(P, 2) = size(MAT, 2) = n .

D (INPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix BD as returned by BD_CMP, BD_CMP2, SELECT_SINGVAL_CMP3 or SELECT_SINGVAL4 subroutines.

The size of D must verify: size(D) = size(MAT, 2) = n .

E (INPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the bidiagonal matrix BD as returned by BD_CMP, BD_CMP2, SELECT_SINGVAL_CMP3 or SELECT_SINGVAL4 subroutines:

E(i) = BD(i-1,i) for i = 2,3,. . . ,n;

E(1) is arbitrary.

The size of E must verify: size(E) = size(MAT, 2) = n .

S (INPUT) real(stnd), dimension(:) On entry, selected singular values of the bidiagonal matrix BD. The singular values must be given in decreasing order and are assumed to be positive or zero.

The size of S must verify: size(S) <= size(MAT, 2) = n .

LEFTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed left singular vectors. The left singular vector associated with the singular value S(j) is stored in the j-th column of LEFTVEC.

The shape of LEFTVEC must verify:

- size(LEFTVEC, 1) = size(MAT, 1) = m ,
- size(LEFTVEC, 2) = size(S) .

RIGHTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed right singular vectors. The right singular vector associated with the singular value S(j) is stored in the j-th column of RIGHTVEC.

The shape of RIGHTVEC must verify:

- size(RIGHTVEC, 1) = size(MAT, 2) = n ,
- size(RIGHTVEC, 2) = size(S) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = FALSE : indicates successful exit,
- FAILURE = TRUE : indicates that some singular vectors of BD failed to converge in MAXITER iterations.

MAXITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed in the subroutine.

By default, 2 inverse iterations are performed for all the singular vectors.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, all the singular vectors of the bidiagonal matrix BD are orthogonalized by the Modified Gram-Schmidt or QR algorithm;
- ORTHO=false, the singular vectors of the bidiagonal matrix BD are not orthogonalized by the Modified Gram-Schmidt or QR algorithm.

The default is to orthogonalize the singular vectors only for the singular values, which are not well-separated.

BACKWARD_SWEEP (INPUT, OPTIONAL) logical(lgl) On entry, if:

- BACKWARD_SWEEP=true and the singular vectors of the bidiagonal matrix BD are orthogonalized by the modified Gram-Schmidt algorithm, a backward sweep of the modified Gram-Schmidt algorithm is also performed;
- BACKWARD_SWEEP=false, a backward sweep is not performed.

The default is not to perform a backward sweep of the modified Gram-Schmidt algorithm.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if:

- SCALING=true, the bidiagonal matrix BD is scaled before computing the singular vectors;
- SCALING=false, the bidiagonal matrix BD is not scaled.

The default is to scale the bidiagonal matrix.

INITVEC (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITVEC=true, Fernando vectors are used to start the inverse iteration process for computing the singular vectors of the bidiagonal matrix BD (e.g. the eigenvectors of the associated Golub-Kahan tridiagonal matrix);
- INITVEC=false, random uniform starting vectors are used.

The default is to use Fernando starting vectors if the singular values are well-separated and the Golub-Kahan form of the input bidiagonal matrix is unreduced, and random uniform starting vectors otherwise.

TOL_REORTHO (INPUT, OPTIONAL) real(stnd) On entry, TOL_REORTHO is used to determine if the left singular vectors stored in LEFTVEC must be reortogonalized on exit in order to correct for the loss of orthogonality in the Ralha-Barlow one-sided bidiagonal reduction algorithm if MAT is nearly deficient. If one of the singular values, S(i), verifies the condition

S(i) <= TOL_REORTHO * S(1)

all the computed left singular vectors are reorthogonalized with a QR factorization. If S(1) is the largest singular value of MAT, this condition leads to the assertion that the rank of MAT is less than size(S) and is thus a nearly singular matrix if TOL_REORTHO is a small positive value of the order of the machine epsilon.

TOL_REORTHO must be greater or equal to zero and less than or equal to one. If TOL_REORTHO = 0. is used, the left singular vectors are reorthogonalized only if some singular values are almost zero. On the other hand, If TOL_REORTHO = 1. is used, the left singular vectors are always reorthogonalized. If TOL_REORTHO is specified as less than zero or greater than one, the default value is used.

The default value is the value of the module parameter tol_reortho_def if size(S) = n and tol_reortho_partial_def otherwise.

Further Details

A first estimate of the singular vectors is computed by the Fernando method applied to the tridiagonal Golub-Kahan matrix associated with the bidiagonal matrix BD (see the reference (1) for details) for the singular values which are well-separated and if the Golub-Kahan form of the input bidiagonal matrix is unreduced. For the other singular values, a random start is used as a first estimate of the singular vectors as in the standard inverse-iteration algorithm.

The singular vectors of BD are then computed or refined using inverse iteration on the tridiagonal Golub-Kahan matrix for all the singular values at one step.

By default, the singular vectors of BD are then orthogonalized by the Modified Gram-Schmidt or QR algorithm only if the singular values are not well-separated.

The singular vectors of MAT are finally computed by a blocked back-transformation algorithm.

The computation of the singular vectors of BD and the blocked back-transformation algorithm to find the singular vectors of MAT are parallelized if OPENMP is used.

BD_INVITER2 may fail if some singular values specified in parameter S are nearly identical for some pathological matrices.

For further details on Fernando method for computing eigenvectors of tridiagonal matrices, the blocked back-transformation algorithm or inverse iteration, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(2) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

(3) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.19.66 subroutine bd_inviter2 (mat, tauq, taup, rlmat, d, e, s, leftvec, rightvec, failure, tauo, maxiter, ortho, backward_sweep, scaling, initvec)

Purpose

BD_INVITER2 computes the left and right singular vectors of a full real m-by-n matrix MAT corresponding to specified singular values, using inverse iteration.

It is required that the original matrix MAT has been reduced to upper bidiagonal form BD by a two-step algorithm as performed by BD_CMP subroutine with parameters TAUQ, TAUP, RLMAT, and eventually TAUO, or more simply by SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 subroutines with the same arguments:

- If m >= n, a QR factorization of the real m-by-n matrix MAT is first computed

  MAT = O * R

where O is orthogonal and R is upper triangular. In a second step, the n-by-n upper triangular matrix R is reduced to upper bidiagonal form BD by an orthogonal transformation:

Q' * R * P = BD

where Q and P are orthogonal and BD is an upper bidiagonal matrix.

- If m < n, an LQ factorization of the real m-by-n matrix MAT is first computed

  MAT = L * O

where O is orthogonal and L is lower triangular. In a second step, the m-by-m lower triangular matrix L is reduced to upper bidiagonal form BD by an orthogonal transformation:

Q' * L * P = BD

where Q and P are orthogonal and BD is an upper bidiagonal matrix.

After this call to BD_CMP with parameters TAUQ, TAUP, RLMAT, and eventually TAUO, the user can call BD_SVD, BD_SINGVAL or BD_SINGVAL2 subroutines for computing singular values of BD and, finally, BD_INVITER2 for computing all or selected singular vectors of MAT.

Alternatively and more simply, the user can call SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 subroutines, with parameters TAUQ, TAUP, RLMAT, and eventually TAUO, to perform the two-stage bidiagonal reduction and get singular values in one step and, finally, call BD_INVITER2 for computing all or selected singular vectors of MAT.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the original m-by-n matrix after a two-stage bidiagonal reduction by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 subroutines (e.g. when the matrix argument RLMAT is present in the call to these subroutines). The precise content of MAT is determined by the presence or absence of the optional argument TAUO:

- if the optional argument TAUO is absent, it is assumed that the first n columns (if m>=n) or the first m rows (if m<n) of the orthogonal matrix O is stored explicitly in the argument MAT on entry;
- if the optional argument TAUO is present, it is assumed that the orthogonal matrix O is stored in factored form, as a product of elementary reflectors, in the argument MAT on entry. In this case, MAT must contains the vectors which define the elementary reflectors W(i) whose products determine the matrix O, as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 subroutines when the matrix argument RLMAT is present in the call to these subroutines.

In both cases, MAT must be specified as returned by BD_CMP and is not modified by the routine.

TAUQ (INPUT) real(stnd), dimension(:) TAUQ(i) must contain the scalar factor of the elementary reflector H(i) which determines Q, as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 in their array argument TAUQ.

The size of TAUQ must verify: size(TAUQ) = min(size(MAT,1) , size(MAT,2)) .

TAUP (INPUT) real(stnd), dimension(:) TAUP(i) must contain the scalar factor of the elementary reflector G(i), which determines P, as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 in their array argument TAUP.

The size of TAUP must verify: size(TAUP) = min(size(MAT,1) , size(MAT,2)) .

RLMAT (INPUT) real(stnd), dimension(:,:) On entry, the elements on and below the diagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array TAUP, represent the orthogonal matrix P as a product of elementary reflectors. RLMAT must be specified as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 and is not modified by the routine.

The shape of RLMAT must verify: size(RLMAT, 1) = size(RLMAT, 2) = min(size(MAT,1) , size(MAT,2)) .

D (INPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the upper bidiagonal matrix BD as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2.

The size of D must verify: size(D) = min(size(MAT,1) , size(MAT,2)) .

E (INPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the upper bidiagonal matrix BD as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2:

E(i) = BD(i-1,i) for i = 2,3,. . . ,min(m,n);

E(1) is arbitrary.

The size of E must verify: size(E) = min(size(MAT,1) , size(MAT,2)) .

S (INPUT) real(stnd), dimension(:) On entry, selected singular values of the upper bidiagonal matrix BD. The singular values must be given in decreasing order and are assumed to be positive or zero.

The size of S must verify: size(S) <= min(size(MAT,1) , size(MAT,2)) .

LEFTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed left singular vectors. The left singular vector associated with the singular value S(j) is stored in the j-th column of LEFTVEC.

The shape of LEFTVEC must verify:

- size(LEFTVEC, 1) = size(MAT, 1) = m ,
- size(LEFTVEC, 2) = size(S) .

RIGHTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed right singular vectors. The right singular vector associated with the singular value S(j) is stored in the j-th column of RIGHTVEC.

The shape of RIGHTVEC must verify:

- size(RIGHTVEC, 1) = size(MAT, 2) = n ,
- size(RIGHTVEC, 2) = size(S) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = FALSE : indicates successful exit ,
- FAILURE = TRUE : indicates that some singular vectors of BD failed to converge in MAXITER iterations .

TAUO (INPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors W(i), which represent the orthogonal matrix O of the QR or LQ decomposition of MAT.

If the optional argument TAUO is present, it is assumed that the orthogonal matrix O is stored in factored form, as a product of elementary reflectors, in the argument MAT on entry.

If the optional argument TAUO is absent, it is assumed that the orthogonal matrix O is stored explicitly in the argument MAT on entry.

If the optional argument TAUO has been specified in the initial call to the BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 subroutines, this optional argument TAUO must also be specified in the call to BD_INVITER2 for computing singular vectors, otherwise the results will be incorrect.

See description of the argument MAT in the description of the BD_CMP subroutine, when the argument RLMAT is also present, for further details.

The size of TAUO must be min(size(MAT,1) , size(MAT,2)).

MAXITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed in the subroutine.

By default, 2 inverse iterations are performed for all the singular vectors.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, all the singular vectors of the bidiagonal matrix BD are orthogonalized by the Modified Gram-Schmidt or QR algorithm;
- ORTHO=false, the singular vectors of the bidiagonal matrix BD are not orthogonalized by the Modified Gram-Schmidt or QR algorithm.

The default is to orthogonalize the singular vectors only for the singular values, which are not well-separated.

BACKWARD_SWEEP (INPUT, OPTIONAL) logical(lgl) On entry, if:

- BACKWARD_SWEEP=true and the singular vectors of the bidiagonal matrix BD are orthogonalized by the modified Gram-Schmidt algorithm, a backward sweep of the modified Gram-Schmidt algorithm is also performed;
- BACKWARD_SWEEP=false, a backward sweep is not performed.

The default is not to perform a backward sweep of the modified Gram-Schmidt algorithm.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if:

- SCALING=true, the bidiagonal matrix BD is scaled before computing the singular vectors;
- SCALING=false, the bidiagonal matrix BD is not scaled.

The default is to scale the bidiagonal matrix.

INITVEC (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITVEC=true, Fernando vectors are used to start the inverse iteration process for computing the singular vectors of the bidiagonal matrix BD (e.g. the eigenvectors of the associated Golub-Kahan tridiagonal matrix);
- INITVEC=false, random uniform starting vectors are used.

The default is to use Fernando starting vectors if the singular values are well-separated and the Golub-Kahan form of the input bidiagonal matrix is unreduced, and random uniform starting vectors otherwise.

Further Details

A first estimate of the singular vectors is computed by the Fernando method applied to the tridiagonal Golub-Kahan matrix associated with the bidiagonal matrix BD (see the reference (1) for details) for the singular values which are well-separated and if the Golub-Kahan form of the input bidiagonal matrix is unreduced. For the other singular values, a random start is used as a first estimate of the singular vectors as in the standard inverse-iteration algorithm.

The singular vectors of BD are then computed or refined using inverse iteration on the tridiagonal Golub-Kahan matrix for all the singular values at one step.

By default, the singular vectors of BD are then orthogonalized by the Modified Gram-Schmidt or QR algorithm only if the singular values are not well-separated.

The singular vectors of MAT are finally computed by a blocked back-transformation algorithm.

The computation of the singular vectors of BD and the blocked back-transformation algorithm to find the singular vectors of MAT are parallelized if OPENMP is used.

BD_INVITER2 may fail if some singular values specified in parameter S are nearly identical for some pathological matrices.

For further details on Fernando method for computing eigenvectors of tridiagonal matrices, the blocked back-transformation algorithm or inverse iteration, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(2) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

(3) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.19.67 subroutine bd_inviter2 (mat, rmat, p, d, e, s, leftvec, rightvec, failure, tauo, maxiter, ortho, backward_sweep, scaling, initvec, tol_reortho)

Purpose

BD_INVITER2 computes all or selected left and right singular vectors of a full real m-by-n matrix MAT with m>=n corresponding to specified singular values, using inverse iteration.

It is required that the original matrix MAT has been reduced to upper bidiagonal form BD by a two-step algorithm as performed by SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 subroutines with parameters P, RMAT, and eventually TAUO:

A QR factorization of the real m-by-n matrix MAT is first computed

MAT = O * R

where O is orthogonal and R is upper triangular. In a second step, the n-by-n upper triangular matrix R is reduced to upper bidiagonal form BD by an orthogonal transformation :

Q' * R * P = BD

where Q and P are orthogonal and BD is an upper bidiagonal matrix. Subroutines SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 compute O, Q, P, BD and also all or some of the singular values of R, which are also the singular values of MAT. Using this two-step factorization, BD_INVITER2 computes all or selected left and right singular vectors of R and apply to them a back-transformation algorithm to obtain the corresponding left and right singular vectors of MAT.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the original m-by-n matrix after the two-step bidiagonal reduction by SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 subroutines with arguments P, RMAT, and eventually TAUO. The precise content of MAT is determined by the presence or absence of the optional argument TAUO:

- if the optional argument TAUO is absent, it is assumed that the first n columns of the orthogonal matrix O is stored explicitly in the argument MAT on entry,
- if the optional argument TAUO is present, it is assumed that the orthogonal matrix O is stored in factored form, as a product of elementary reflectors, in the argument MAT on entry. In this case, MAT must contains the vectors which define the elementary reflectors W(i) whose products determine the matrix O, as returned by SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 subroutines when the matrix argument RLMAT is present in the call to these subroutines.

In both cases, MAT must be specified as returned by these subroutines and is not modified by the routine. See the description of the argument TAUO below for further details.

The shape of MAT must verify: size(MAT, 1) >= size(MAT, 2) = n .

RMAT (INPUT) real(stnd), dimension(:,:) On entry, the n-by-n orthogonal matrix Q after reduction by SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 subroutines. RMAT must be specified as returned by these subroutines and is not modified by the routine.

The shape of RMAT must verify: size(RMAT, 1) = size(RMAT, 2) = size(MAT, 2) = n.

P (INPUT) real(stnd), dimension(:,:) On entry, the n-by-n orthogonal matrix P after reduction by SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 subroutines. P can be stored in factored form or not. Both cases are handled by the subroutine and P is not modified by the routine.

The shape of P must verify: size(P, 1) = size(P, 2) = size(MAT, 2) = n .

D (INPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix BD as returned by SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4.

The size of D must verify: size(D) = size(MAT, 2) = n .

E (INPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the bidiagonal matrix BD as returned by SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4:

E(i) = BD(i-1,i) for i = 2,3,. . . ,n;

E(1) is arbitrary.

The size of E must verify: size(E) = size(MAT, 2) = n .

S (INPUT) real(stnd), dimension(:) On entry, selected singular values of the bidiagonal matrix BD. The singular values must be given in decreasing order and are assumed to be positive or zero.

The size of S must verify: size(S) <= size(MAT, 2) = n .

LEFTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed left singular vectors. The left singular vector associated with the singular value S(j) is stored in the j-th column of LEFTVEC.

The shape of LEFTVEC must verify:

- size(LEFTVEC, 1) = size(MAT, 1) = m ,
- size(LEFTVEC, 2) = size(S) .

RIGHTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed right singular vectors. The right singular vector associated with the singular value S(j) is stored in the j-th column of RIGHTVEC.

The shape of RIGHTVEC must verify:

- size(RIGHTVEC, 1) = size(MAT, 2) = n ,
- size(RIGHTVEC, 2) = size(S) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = FALSE : indicates successful exit,
- FAILURE = TRUE : indicates that some singular vectors of BD failed to converge in MAXITER iterations.

TAUO (INPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors W(i), which represent the orthogonal matrix O of the QR decomposition of MAT as returned by SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4.

If the optional argument TAUO is present, it is assumed that the orthogonal matrix O is stored in factored form, as a product of elementary reflectors, in the argument MAT on entry.

If the optional argument TAUO is absent, it is assumed that the orthogonal matrix O is stored explicitly in the argument MAT on entry.

If the optional argument TAUO has been specified in the initial call to the SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 subroutines, this optional argument TAUO must also be specified in the call to BD_INVITER2, otherwise the results will be incorrect.

See description of the argument MAT in the description of the SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 subroutines, when the argument RMAT is also present, for further details.

The size of TAUO must be size(MAT,2) = n .

MAXITER (INPUT, OPTIONAL) integer(i4b) The number of inverse iterations performed in the subroutine.

By default, 2 inverse iterations are performed for all the singular vectors.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, all the singular vectors of the bidiagonal matrix BD are orthogonalized by the Modified Gram-Schmidt or QR algorithm;
- ORTHO=false, the singular vectors of the bidiagonal matrix BD are not orthogonalized by the Modified Gram-Schmidt or QR algorithm.

The default is to orthogonalize the singular vectors only for the singular values, which are not well-separated.

BACKWARD_SWEEP (INPUT, OPTIONAL) logical(lgl) On entry, if:

- BACKWARD_SWEEP=true and the singular vectors of the bidiagonal matrix BD are orthogonalized by the modified Gram-Schmidt algorithm, a backward sweep of the modified Gram-Schmidt algorithm is also performed;
- BACKWARD_SWEEP=false, a backward sweep is not performed.

The default is not to perform a backward sweep of the modified Gram-Schmidt algorithm.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if:

- SCALING=true, the bidiagonal matrix BD is scaled before computing the singular vectors;
- SCALING=false, the bidiagonal matrix BD is not scaled.

The default is to scale the bidiagonal matrix.

INITVEC (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITVEC=true, Fernando vectors are used to start the inverse iteration process for computing the singular vectors of the bidiagonal matrix BD (e.g. the eigenvectors of the associated Golub-Kahan tridiagonal matrix);
- INITVEC=false, random uniform starting vectors are used.

The default is to use Fernando starting vectors if the singular values are well-separated and the Golub-Kahan form of the input bidiagonal matrix is unreduced, and random uniform starting vectors otherwise.

TOL_REORTHO (INPUT, OPTIONAL) real(stnd) On entry, TOL_REORTHO is used to determine if the left singular vectors stored in LEFTVEC must be reortogonalized on exit in order to correct for the loss of orthogonality in the Ralha-Barlow one-sided bidiagonal reduction algorithm if MAT is nearly deficient. If one of the singular values, S(i), verifies the condition

S(i) <= TOL_REORTHO * S(1)

all the computed left singular vectors are reorthogonalized with a QR factorization. If S(1) is the largest singular value of MAT, this condition leads to the assertion that the rank of MAT is less than size(S) and is thus a nearly singular matrix if TOL_REORTHO is a small positive value of the order of the machine epsilon.

TOL_REORTHO must be greater or equal to zero and less than or equal to one. If TOL_REORTHO = 0. is used, the left singular vectors are reorthogonalized only if some singular values are almost zero. On the other hand, If TOL_REORTHO = 1. is used, the left singular vectors are always reorthogonalized. If TOL_REORTHO is specified as less than zero or greater than one, the default value is used.

The default value is the value of the module parameter tol_reortho_def if size(S) = n and tol_reortho_partial_def otherwise.

Further Details

A first estimate of the singular vectors is computed by the Fernando method applied to the tridiagonal Golub-Kahan matrix associated with the bidiagonal matrix BD (see the reference (1) for details) for the singular values which are well-separated and if the Golub-Kahan form of the input bidiagonal matrix is unreduced. For the other singular values, a random start is used as a first estimate of the singular vectors as in the standard inverse-iteration algorithm.

The singular vectors of BD are then computed or refined using inverse iteration on the tridiagonal Golub-Kahan matrix for all the singular values at one step.

By default, the singular vectors of BD are then orthogonalized by the Modified Gram-Schmidt or QR algorithm only if the singular values are not well-separated.

The singular vectors of MAT are finally computed by a blocked back-transformation algorithm.

The computation of the singular vectors of BD and the blocked back-transformation algorithm to find the singular vectors of MAT are parallelized if OPENMP is used.

BD_INVITER2 may fail if some singular values specified in parameter S are nearly identical for some pathological matrices.

For further details on Fernando method for computing eigenvectors of tridiagonal matrices, the blocked back-transformation algorithm or inverse iteration, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(2) **Parlett, B.N., and Dhillon, I.S., 1997:** Fernando's solution to Wilkinson's problem: An application of double factorization. Linear Algebra and its Appl., 267, pp.247-279.

(3) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.19.68 `subroutine upper_bd_dsqd2 ( q2, e2, shift, flip, d )`

Purpose

UPPER_BD_DSQD2 computes:

- the L * D * L' factorization of the matrix BD' * BD - shift * I , if FLIP=false;
- the U * D * U' factorization of the matrix BD * BD' - shift * I , if FLIP=true;

for a n-by-n (upper) bidiagonal matrix BD and a given shift. L and U are, respectively, unit lower and unit upper bidiagonal matrices and D is a diagonal matrix.

The differential form of the stationary QD algorithm of Rutishauser is used to compute the factorization from the squared elements of the bidiagonal matrix BD (see the reference (1) below for further details).

The subroutine outputs the diagonal matrix D of the factorization.

Arguments

Q2 (INPUT) real(stnd), dimension(:) On entry, Q2 contains the squared diagonal elements of the bidiagonal matrix BD.

E2 (INPUT) real(stnd), dimension(:) On entry, the n-1 squared off-diagonal elements of the bidiagonal matrix BD.

The size of E2 must be size(E2) = size(Q2) - 1.

SHIFT (INPUT) real(stnd) On entry, the shift.

FLIP (INPUT) logical(lgl) On entry, if FLIP=false the L * D * L' factorization of the matrix BD' * BD - shift * I is computed. Otherwise, if FLIP=true the U * D * U' factorization of the matrix BD * BD' - shift * I is computed.

D (OUTPUT) real(stnd), dimension(:) On exit, the elements of the diagonal matrix D.

The size of D must be size(D) = size(Q2).

Further Details

The bidiagonal matrix BD must be scaled appropriately before using this subroutine in order to avoid overflows (see the reference (1) below for further details).

This subroutine is adapted from the algorithms given in reference (1). See:

(1) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

6.19.69 subroutine upper_bd_dpqd2 (q2, e2, shift, flip, d)

Purpose

UPPER_BD_DPQD2 computes:

- the L * D * L' factorization of the matrix BD * BD' - shift * I , if FLIP=false;
- the U * D * U' factorization of the matrix BD' * BD - shift * I , if FLIP=true;

for a n-by-n (upper) bidiagonal matrix BD and a given shift. L and U are, respectively, unit lower and unit upper bidiagonal matrices and D is a diagonal matrix.

The differential form of the progressive QD algorithm of Rutishauser is used to compute the factorization from the squared elements of the bidiagonal matrix BD (see the reference (1) below for further details).

The subroutine outputs the diagonal matrix D of the factorization.

Arguments

Q2 (INPUT) real(stnd), dimension(:) On entry, Q2 contains the squared diagonal elements of the bidiagonal matrix BD.

E2 (INPUT) real(stnd), dimension(:) On entry, the n-1 squared off-diagonal elements of the bidiagonal matrix BD.

The size of E2 must be size(E2) = size(Q2) - 1.

SHIFT (INPUT) real(stnd) On entry, the shift.

FLIP (INPUT) logical(lgl) On entry, if FLIP=false the L * D * L' factorization of the matrix BD * BD' - shift * I is computed. Otherwise, if FLIP=true the U * D * U' factorization of the matrix BD' * BD - shift * I is computed.

D (OUTPUT) real(stnd), dimension(:) On exit, the elements of the diagonal matrix D.

The size of D must be size(D) = size(Q2).

Further Details

The bidiagonal matrix BD must be scaled appropriately before using this subroutine in order to avoid overflows (see the reference (1) below for further details).

This subroutine is adapted from the algorithms given in reference (1). See:

(1) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

6.19.70 `subroutine upper_bd_dsqd2 ( q2, e2, shift, flip, d, t )`

Purpose

UPPER_BD_DSQD2 computes:

- the L * D * L' factorization of the matrix BD' * BD - shift * I , if FLIP=false;
- the U * D * U' factorization of the matrix BD * BD' - shift * I , if FLIP=true;

for a n-by-n (upper) bidiagonal matrix BD and a given shift. L and U are, respectively, unit lower and unit upper bidiagonal matrices and D is a diagonal matrix.

The differential form of the stationary QD algorithm of Rutishauser is used to compute the factorization from the squared elements of the bidiagonal matrix BD (see the reference (1) below for further details).

The subroutine outputs the diagonal matrix D of the factorization and the auxiliary variable T in the differential form of the stationary QD algorithm.

Arguments

Q2 (INPUT) real(stnd), dimension(:) On entry, Q2 contains the squared diagonal elements of the bidiagonal matrix BD.

E2 (INPUT) real(stnd), dimension(:) On entry, the n-1 squared off-diagonal elements of the bidiagonal matrix BD.

The size of E2 must be size(E2) = size(Q2) - 1.

SHIFT (INPUT) real(stnd) On entry, the shift.

FLIP (INPUT) logical(lgl) On entry, if FLIP=false the L * D * L' factorization of the matrix BD' * BD - shift * I is computed. Otherwise, if FLIP=true the U * D * U' factorization of the matrix BD * BD' - shift * I is computed.

D (OUTPUT) real(stnd), dimension(:) On exit, the elements of the diagonal matrix D.

The size of D must be size(D) = size(Q2).

T (OUTPUT) real(stnd), dimension(:) On exit, the vector of the auxiliary values T(i) in the differential form of the stationary QD algorithm.

The size of T must be size(T) = size(D) = size(Q2).

Further Details

The bidiagonal matrix BD must be scaled appropriately before using this subroutine in order to avoid overflows (see the reference (1) below for further details).

This subroutine is adapted from the algorithms given in reference (1). See:

(1) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

6.19.71 `subroutine upper_bd_dpqd2 ( q2, e2, shift, flip, d, s )`

Purpose

UPPER_BD_DPQD2 computes:

- the L * D * L' factorization of the matrix BD * BD' - shift * I , if FLIP=false;
- the U * D * U' factorization of the matrix BD' * BD - shift * I , if FLIP=true;

for a n-by-n (upper) bidiagonal matrix BD and a given shift. L and U are, respectively, unit lower and unit upper bidiagonal matrices and D is a diagonal matrix.

The differential form of the progressive QD algorithm of Rutishauser is used to compute the factorization from the squared elements of the bidiagonal matrix BD (see the reference (1) below for further details).

The subroutine outputs the diagonal matrix D of the factorization and the auxiliary variable S in the differential form of the progressive QD algorithm.

Arguments

Q2 (INPUT) real(stnd), dimension(:) On entry, Q2 contains the squared diagonal elements of the bidiagonal matrix BD.

E2 (INPUT) real(stnd), dimension(:) On entry, the n-1 squared off-diagonal elements of the bidiagonal matrix BD.

The size of E2 must be size(E2) = size(Q2) - 1.

SHIFT (INPUT) real(stnd) On entry, the shift.

FLIP (INPUT) logical(lgl) On entry, if FLIP=false the L * D * L' factorization of the matrix BD * BD' - shift * I is computed. Otherwise, if FLIP=true the U * D * U' factorization of the matrix BD' * BD - shift * I is computed.

D (OUTPUT) real(stnd), dimension(:) On exit, the elements of the diagonal matrix D.

The size of D must be size(D) = size(Q2).

S (OUTPUT) real(stnd), dimension(:) On exit, the vector of the auxiliary values S(i) in the differential form of the progressive QD algorithm.

The size of S must be size(S) = size(D) = size(Q2).

Further Details

The bidiagonal matrix BD must be scaled appropriately before using this subroutine in order to avoid overflows (see the reference (1) below for further details).

This subroutine is adapted from the algorithms given in reference (1). See:

(1) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

6.19.72 `subroutine upper_bd_dsqd ( a, b, shift, flip, d )`

Purpose

UPPER_BD_DSQD computes:

- the L * D * L' factorization of the matrix BD' * BD - shift * I , if FLIP=false;
- the U * D * U' factorization of the matrix BD * BD' - shift * I , if FLIP=true;

for a n-by-n (upper) bidiagonal matrix BD and a given shift. L and U are, respectively, unit lower and unit upper bidiagonal matrices and D is a diagonal matrix.

The differential form of the stationary QD algorithm of Rutishauser is used to compute the factorization (see the reference (1) below for further details).

The subroutine outputs the diagonal matrix D of the factorization.

Arguments

A (INPUT) real(stnd), dimension(:) On entry, A contains the diagonal elements of the bidiagonal matrix BD.

B (INPUT) real(stnd), dimension(:) On entry, the n-1 off-diagonal elements of the bidiagonal matrix BD.

The size of B must be size(B) = size(A) - 1.

SHIFT (INPUT) real(stnd) On entry, the shift.

FLIP (INPUT) logical(lgl) On entry, if FLIP=false the L * D * L' factorization of the matrix BD' * BD - shift * I is computed. Otherwise, if FLIP=true the U * D * U' factorization of the matrix BD * BD' - shift * I is computed.

D (OUTPUT) real(stnd), dimension(:) On exit, the elements of the diagonal matrix D.

The size of D must be size(D) = size(A).

Further Details

The bidiagonal matrix BD must be scaled appropriately before using this subroutine in order to avoid overflows (see the reference (1) below for further details).

This subroutine is adapted from the algorithms given in reference (1). See:

(1) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

6.19.73 subroutine upper_bd_dpqd (a, b, shift, flip, d)

Purpose

UPPER_BD_DPQD computes:

- the L * D * L' factorization of the matrix BD * BD' - shift * I , if FLIP=false;
- the U * D * U' factorization of the matrix BD' * BD - shift * I , if FLIP=true;

for a n-by-n (upper) bidiagonal matrix BD and a given shift. L and U are, respectively, unit lower and unit upper bidiagonal matrices and D is a diagonal matrix.

The differential form of the progressive QD algorithm of Rutishauser is used to compute the factorization (see the reference (1) below for further details).

The subroutine outputs the diagonal matrix D of the factorization.

Arguments

A (INPUT) real(stnd), dimension(:) On entry, A contains the diagonal elements of the bidiagonal matrix BD.

B (INPUT) real(stnd), dimension(:) On entry, the n-1 off-diagonal elements of the bidiagonal matrix BD.

The size of B must be size(B) = size(A) - 1.

SHIFT (INPUT) real(stnd) On entry, the shift.

FLIP (INPUT) logical(lgl) On entry, if FLIP=false the L * D * L' factorization of the matrix BD * BD' - shift * I is computed. Otherwise, if FLIP=true the U * D * U' factorization of the matrix BD' * BD - shift * I is computed.

D (OUTPUT) real(stnd), dimension(:) On exit, the elements of the diagonal matrix D.

The size of D must be size(D) = size(A).

Further Details

The bidiagonal matrix BD must be scaled appropriately before using this subroutine in order to avoid overflows (see the reference (1) below for further details).

This subroutine is adapted from the algorithms given in reference (1). See:

(1) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

6.19.74 subroutine upper_bd_dsqd (a, b, shift, flip, d, t)

Purpose

UPPER_BD_DSQD computes:

- the L * D * L' factorization of the matrix BD' * BD - shift * I , if FLIP=false;
- the U * D * U' factorization of the matrix BD * BD' - shift * I , if FLIP=true;

for a n-by-n (upper) bidiagonal matrix BD and a given shift. L and U are, respectively, unit lower and unit upper bidiagonal matrices and D is a diagonal matrix.

The differential form of the stationary QD algorithm of Rutishauser is used to compute the factorization (see the reference (1) below for further details).

The subroutine outputs the diagonal matrix D of the factorization and the auxiliary variable T in the differential form of the stationary QD algorithm.

Arguments

A (INPUT) real(stnd), dimension(:) On entry, A contains the diagonal elements of the bidiagonal matrix BD.

B (INPUT) real(stnd), dimension(:) On entry, the n-1 off-diagonal elements of the bidiagonal matrix BD.

The size of B must be size(B) = size(A) - 1.

SHIFT (INPUT) real(stnd) On entry, the shift.

FLIP (INPUT) logical(lgl) On entry, if FLIP=false the L * D * L' factorization of the matrix BD' * BD - shift * I is computed. Otherwise, if FLIP=true the U * D * U' factorization of the matrix BD * BD' - shift * I is computed.

D (OUTPUT) real(stnd), dimension(:) On exit, the elements of the diagonal matrix D.

The size of D must be size(D) = size(A).

T (OUTPUT) real(stnd), dimension(:) On exit, the vector of the auxiliary values T(i) in the differential form of the stationary QD algorithm.

The size of T must be size(T) = size(D) = size(A).

Further Details

The bidiagonal matrix BD must be scaled appropriately before using this subroutine in order to avoid overflows (see the reference (1) below for further details).

This subroutine is adapted from the algorithms given in reference (1). See:

(1) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

6.19.75 `subroutine upper_bd_dpqd ( a, b, shift, flip, d, s )`

Purpose

UPPER_BD_DPQD computes:

- the L * D * L' factorization of the matrix BD * BD' - shift * I , if FLIP=false;
- the U * D * U' factorization of the matrix BD' * BD - shift * I , if FLIP=true;

for a n-by-n (upper) bidiagonal matrix BD and a given shift. L and U are, respectively, unit lower and unit upper bidiagonal matrices and D is a diagonal matrix.

The differential form of the progressive QD algorithm of Rutishauser is used to compute the factorization (see the reference (1) below for further details).

The subroutine outputs the diagonal matrix D of the factorization and the auxiliary variable S in the differential form of the progressive QD algorithm.

Arguments

A (INPUT) real(stnd), dimension(:) On entry, A contains the diagonal elements of the bidiagonal matrix BD.

B (INPUT) real(stnd), dimension(:) On entry, the n-1 off-diagonal elements of the bidiagonal matrix BD.

The size of B must be size(B) = size(A) - 1.

SHIFT (INPUT) real(stnd) On entry, the shift.

FLIP (INPUT) logical(lgl) On entry, if FLIP=false the L * D * L' factorization of the matrix BD * BD' - shift * I is computed. Otherwise, if FLIP=true the U * D * U' factorization of the matrix BD' * BD - shift * I is computed.

D (OUTPUT) real(stnd), dimension(:) On exit, the elements of the diagonal matrix D.

The size of D must be size(D) = size(A).

S (OUTPUT) real(stnd), dimension(:) On exit, the vector of the auxiliary values S(i) in the differential form of the progressive QD algorithm.

The size of S must be size(S) = size(D) = size(A).

Further Details

The bidiagonal matrix BD must be scaled appropriately before using this subroutine in order to avoid overflows (see the reference (1) below for further details).

This subroutine is adapted from the algorithms given in reference (1). See:

(1) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

6.19.76 `subroutine upper_bd_dsqd ( a, b, shift, flip, d, t, l )`

Purpose

UPPER_BD_DSQD computes:

- the L * D * L' factorization of the matrix BD' * BD - shift * I , if FLIP=false;
- the U * D * U' factorization of the matrix BD * BD' - shift * I , if FLIP=true;

for a n-by-n (upper) bidiagonal matrix BD and a given shift. L and U are, respectively, unit lower and unit upper bidiagonal matrices and D is a diagonal matrix.

The differential form of the stationary QD algorithm of Rutishauser is used to compute the factorization (see the reference (1) below for further details).

The subroutine outputs the diagonal matrix D of the factorization, the off-diagonal entries of L (or of U if FLIP=true) and the auxiliary variable T in the differential form of the stationary QD algorithm.

Arguments

A (INPUT) real(stnd), dimension(:) On entry, A contains the diagonal elements of the bidiagonal matrix BD.

B (INPUT) real(stnd), dimension(:) On entry, the n-1 off-diagonal elements of the bidiagonal matrix BD.

The size of B must be size(B) = size(A) - 1.

SHIFT (INPUT) real(stnd) On entry, the shift.

FLIP (INPUT) logical(lgl) On entry, if FLIP=false the L * D * L' factorization of the matrix BD' * BD - shift * I is computed. Otherwise, if FLIP=true the U * D * U' factorization of the matrix BD * BD' - shift * I is computed.

D (OUTPUT) real(stnd), dimension(:) On exit, the elements of the diagonal matrix D.

The size of D must be size(D) = size(A).

T (OUTPUT) real(stnd), dimension(:) On exit, the vector of the auxiliary values T(i) in the differential form of the stationary QD algorithm.

The size of T must be size(T) = size(D) = size(A).

L (OUTPUT) real(stnd), dimension(:) On exit, the off-diagonal entries of L if FLIP=false or the off-diagonal entries of U if FLIP=true.

The size of L must be size(L) = size(B) = size(A) - 1.

Further Details

The bidiagonal matrix BD must be scaled appropriately before using this subroutine in order to avoid overflows (see the reference (1) below for further details).

This subroutine is adapted from the algorithms given in reference (1). See:

(1) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

6.19.77 `subroutine upper_bd_dpqd ( a, b, shift, flip, d, s, l )`

Purpose

UPPER_BD_DPQD computes:

- the L * D * L' factorization of the matrix BD * BD' - shift * I , if FLIP=false;
- the U * D * U' factorization of the matrix BD' * BD - shift * I , if FLIP=true;

for a n-by-n (upper) bidiagonal matrix BD and a given shift. L and U are, respectively, unit lower and unit upper bidiagonal matrices and D is a diagonal matrix.

The differential form of the progressive QD algorithm of Rutishauser is used to compute the factorization (see the reference (1) below for further details).

The subroutine outputs the diagonal matrix D of the factorization, the off-diagonal entries of L (or of U if FLIP=true) and the auxiliary variable S in the differential form of the progressive QD algorithm.

Arguments

A (INPUT) real(stnd), dimension(:) On entry, A contains the diagonal elements of the bidiagonal matrix BD.

B (INPUT) real(stnd), dimension(:) On entry, the n-1 off-diagonal elements of the bidiagonal matrix BD.

The size of B must be size(B) = size(A) - 1.

SHIFT (INPUT) real(stnd) On entry, the shift.

FLIP (INPUT) logical(lgl) On entry, if FLIP=false the L * D * L' factorization of the matrix BD * BD' - shift * I is computed. Otherwise, if FLIP=true the U * D * U' factorization of the matrix BD' * BD - shift * I is computed.

D (OUTPUT) real(stnd), dimension(:) On exit, the elements of the diagonal matrix D.

The size of D must be size(D) = size(A).

S (OUTPUT) real(stnd), dimension(:) On exit, the vector of the auxiliary values S(i) in the differential form of the progressive QD algorithm.

The size of S must be size(S) = size(D) = size(A).

L (OUTPUT) real(stnd), dimension(:) On exit, the off-diagonal entries of L if FLIP=false or the off-diagonal entries of U if FLIP=true.

The size of L must be size(L) = size(B) = size(A) - 1.

Further Details

The bidiagonal matrix BD must be scaled appropriately before using this subroutine in order to avoid overflows (see the reference (1) below for further details).

This subroutine is adapted from the algorithms given in reference (1). See:

(1) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

6.19.78 subroutine dflgen_bd (d, e, lambda, cs_left, sn_left, cs_right, sn_right, scaling)

Purpose

DFLGEN_BD computes deflation parameters (e.g. two chains of Givens rotations) for a n-by-n (upper) bidiagonal matrix BD and a given singular value of BD.

On output, the arguments CS_LEFT, SN_LEFT, CS_RIGHT and SN_RIGHT contain, respectively, the vectors of the cosines and sines coefficients of the chain of n-1 planar rotations that deflates the real n-by-n bidiagonal matrix BD corresponding to a singular value LAMBDA.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix BD.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 off-diagonal elements of the bidiagonal matrix BD.

The size of E must be size(E) = size(D) - 1.

LAMBDA (INPUT) real(stnd) On entry, a singular value of the bidiagonal matrix BD.

CS_LEFT (OUTPUT) real(stnd), dimension(:) On exit, the vector of the cosines coefficients of the chain of n-1 Givens rotations that deflates the bidiagonal matrix BD on the left.

The size of CS_LEFT must be size(CS_LEFT) = size(E) = size(D) - 1.

SN_LEFT (OUTPUT) real(stnd), dimension(:) On exit, the vector of the sines coefficients of the chain of n-1 Givens rotations that deflates the bidiagonal matrix BD on the left.

The size of SN_LEFT must be size(SN_LEFT) = size(E) = size(D) - 1.

CS_RIGHT (OUTPUT) real(stnd), dimension(:) On exit, the vector of the cosines coefficients of the chain of n-1 Givens rotations that deflates the bidiagonal matrix BD on the right.

The size of CS_RIGHT must be size(CS_RIGHT) = size(E) = size(D) - 1.

SN_RIGHT (OUTPUT) real(stnd), dimension(:) On exit, the vector of the sines coefficients of the chain of n-1 Givens rotations that deflates the bidiagonal matrix BD on the right.

The size of SN_RIGHT must be size(SN_RIGHT) = size(E) = size(D) - 1.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the bidiagonal matrix BD is scaled before computing the deflation parameters in order to avoid overflows.

The default is to scale the bidiagonal matrix.

Further Details

This subroutine is adapted from the matlab routine DFLGEN in the reference (1) and algorithms given in reference (2).

For further details, see:

(1) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

(2) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

6.19.79 subroutine dflgen2_bd (d, e, lambda, cs_left, sn_left, cs_right, sn_right, deflate, scaling)

Purpose

DFLGEN2_BD computes and applies deflation parameters (e.g. two chains of Givens rotations) for a n-by-n (upper) bidiagonal matrix BD and a given singular value of BD.

On input:

The arguments D and E contain, respectively, the main diagonal and off-diagonal of the bidiagonal matrix, and the argument LAMBDA contains an estimate of the singular value.

On output:

The arguments D and E contain, respectively, the new main diagonal and off-diagonal of the deflated bidiagonal matrix if DEFLATE is set to true, otherwise D and E are not changed.

The arguments CS_LEFT, SN_LEFT, CS_RIGHT and SN_RIGHT contain, respectively, the vectors of the cosines and sines coefficients of the chain of n-1 planar rotations that deflates the real n-by-n bidiagonal matrix BD corresponding to the singular value LAMBDA. One chain is applied to the left of BD (CS_LEFT, SN_LEFT) and the other is applied to the right of BD (CS_RIGHT, SN_RIGHT).

Arguments

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix BD.

On exit, the new main diagonal of the bidiagonal matrix if DEFLATE=true. Otherwise, D is not changed.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the n-1 off-diagonal elements of the bidiagonal matrix BD.

On exit, the new off-diagonal of the bidiagonal matrix if DEFLATE=true. Otherwise, E is not changed.

The size of E must be size(E) = size(D) - 1.

LAMBDA (INPUT) real(stnd) On entry, a singular value of the bidiagonal matrix BD.

CS_LEFT (OUTPUT) real(stnd), dimension(:) On exit, the vector of the cosines coefficients of the chain of n-1 Givens rotations that deflates the bidiagonal matrix BD on the left.

The size of CS_LEFT must be size(CS_LEFT) = size(E) = size(D) - 1.

SN_LEFT (OUTPUT) real(stnd), dimension(:) On exit, the vector of the sines coefficients of the chain of n-1 Givens rotations that deflates the bidiagonal matrix BD on the left.

The size of SN_LEFT must be size(SN_LEFT) = size(E) = size(D) - 1.

CS_RIGHT (OUTPUT) real(stnd), dimension(:) On exit, the vector of the cosines coefficients of the chain of n-1 Givens rotations that deflates the bidiagonal matrix BD on the right.

The size of CS_RIGHT must be size(CS_RIGHT) = size(E) = size(D) - 1.

SN_RIGHT (OUTPUT) real(stnd), dimension(:) On exit, the vector of the sines coefficients of the chain of n-1 Givens rotations that deflates the bidiagonal matrix BD on the right.

The size of SN_RIGHT must be size(SN_RIGHT) = size(E) = size(D) - 1.

DEFLATE (OUTPUT) logical(lgl) On exit:

- DEFLATE = true : indicates successful exit.
- DEFLATE = false: indicates that full accuracy was not attained in the deflation of the bidiagonal matrix.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the bidiagonal matrix BD is scaled before computing the deflation parameters in order to avoid overflows.

The default is to scale the bidiagonal matrix.

Further Details

This subroutine is adapted from the matlab routine DFLGEN in the reference (1) and algorithms given in reference (2).

For further details, see:

(1) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

(2) **Fernando, K.V., 1998:** Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices. SIAM J. Matrix Anal. Appl., Vol. 20, no 2, pp.373-399.

6.19.80 subroutine dflapp_bd (d, e, cs_left, sn_left, cs_right, sn_right, deflate)

Purpose

DFLAPP_BD deflates a real n-by-n (upper) bidiagonal matrix BD by two chains of planar rotations produced by DFLGEN_BD or DFLGEN2_BD.

On entry, the arguments D and E contain, respectively, the main diagonal and off-diagonal of the bidiagonal matrix.

On output, the arguments D and E contain, respectively, the new main diagonal and off-diagonal of the deflated bidiagonal matrix if DEFLATE is set to true.

Arguments

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix BD.

On exit, the new main diagonal of the bidiagonal matrix if DEFLATE=true. Otherwise, D is not changed.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the n-1 off-diagonal elements of the bidiagonal matrix BD.

On exit, the new off-diagonal of the bidiagonal matrix if DEFLATE=true. Otherwise, E is not changed.

The size of E must be size(E) = size(D) - 1.

CS_LEFT (INPUT) real(stnd), dimension(:) On entry, the vector of the cosines coefficients of the chain of n-1 Givens rotations that deflates the bidiagonal matrix BD on the left.

The size of CS_LEFT must be size(CS_LEFT) = size(E) = size(D) - 1.

SN_LEFT (INPUT) real(stnd), dimension(:) On entry, the vector of the sines coefficients of the chain of n-1 Givens rotations that deflates the bidiagonal matrix BD on the left.

The size of SN_LEFT must be size(SN_LEFT) = size(E) = size(D) - 1.

CS_RIGHT (INPUT) real(stnd), dimension(:) On entry, the vector of the cosines coefficients of the chain of n-1 Givens rotations that deflates the bidiagonal matrix BD on the right.

The size of CS_RIGHT must be size(CS_RIGHT) = size(E) = size(D) - 1.

SN_RIGHT (INPUT) real(stnd), dimension(:) On entry, the vector of the sines coefficients of the chain of n-1 Givens rotations that deflates the bidiagonal matrix BD on the right.

The size of SN_RIGHT must be size(SN_RIGHT) = size(E) = size(D) - 1.

DEFLATE (OUTPUT) logical(lgl) On exit:

- DEFLATE = true : indicates successful exit.
- DEFLATE = false: indicates that full accuracy was not attained in the deflation of the bidiagonal matrix.

Further Details

For further details, see:

(1) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

(2) **Dhillon, I.S., 1998:** Reliable computation of the condition number of a tridiagonal matrix in O(n) time. SIAM J. MATRIX ANAL. APPL, Vol. 19, 776-796.

6.19.81 subroutine qrstep_bd (d, e, lambda, cs_left, sn_left, cs_right, sn_right, deflate, update_bd)

Purpose

QRSTEP_BD performs one QR step with a given shift LAMBDA on a n-by-n real (upper) bidiagonal matrix BD.

On entry, the arguments D and E contain, respectively, the main diagonal and superdiagonal of the bidiagonal matrix.

On output, the arguments D and E contain, respectively, the new main diagonal and superdiagonal of the updated (e.g. deflated) bidiagonal matrix, if DEFLATE is set to true or if the optional logical argument UPDATE_BD is used with the value true, otherwise they are not changed.

The two chains of n-1 planar rotations produced during the QR step with shift LAMBDA are saved in the arguments CS_LEFT, SN_LEFT, CS_RIGHT, SN_RIGHT.

Arguments

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix BD.

On exit, the new main diagonal of the bidiagonal matrix if DEFLATE=true or if UPDATE_BD=true. Otherwise, D is not changed.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the n-1 superdiagonal elements of the bidiagonal matrix BD.

On exit, the new superdiagonal of the bidiagonal matrix if DEFLATE=true or if UPDATE_BD=true. Otherwise, E is not changed.

The size of E must be size(E) = size(D) - 1.

LAMBDA (INPUT) real(stnd) On entry, the shift used in the current QR step.

CS_LEFT (OUTPUT) real(stnd), dimension(:) On exit, the vector of the cosines coefficients of the chain of n-1 Givens rotations applied to the bidiagonal matrix BD on the left in the current QR step.

The size of CS_LEFT must be size(CS_LEFT) = size(E) = size(D) - 1.

SN_LEFT (OUTPUT) real(stnd), dimension(:) On exit, the vector of the sines coefficients of the chain of n-1 Givens rotations applied to the bidiagonal matrix BD on the left in the current QR step.

The size of SN_LEFT must be size(SN_LEFT) = size(E) = size(D) - 1.

CS_RIGHT (OUTPUT) real(stnd), dimension(:) On exit, the vector of the cosines coefficients of the chain of n-1 Givens rotations applied to the bidiagonal matrix BD on the right in the current QR step.

The size of CS_RIGHT must be size(CS_RIGHT) = size(E) = size(D) - 1.

SN_RIGHT (OUTPUT) real(stnd), dimension(:) On exit, the vector of the sines coefficients of the chain of n-1 Givens rotations applied to the bidiagonal matrix BD on the right in the current QR step.

The size of SN_RIGHT must be size(SN_RIGHT) = size(E) = size(D) - 1.

DEFLATE (OUTPUT) logical(lgl) On exit:

- DEFLATE = true : indicates that deflation occured at the end of the step.
- DEFLATE = false: indicates that the last superdiagonal element of the bidiagonal matrix is not small.

UPDATE_BD (INPUT, OPTIONAL) logical(lgl) On entry:

- UPDATE_BD = true : indicates that the bidiagonal matrix will be updated on exit.
- UPDATE_BD = false: indicates that the bidiagonal matrix will be updated on exit only if DEFLATE = true.

The default value for UPDATE_BD is false.

Further Details

This subroutine is adapted from the matlab routine QRSTEP given in the reference (1). The bidiagonal matrix BD is assumed to be unreduced, but no checks are done in the subroutine to verify this hypothesis.

For further details, see:

(1) **Mastronardi, M., Van Barel, M., Van Camp, E., and Vandebril, R., 2006:** On computing the eigenvectors of a class of structured matrices. Journal of Computational and Applied Mathematics, 189, 580-591.

(2) **Demmel, J.W., and Kahan, W., 1990:** Accurate singular values of bidiagonal matrices. SIAM J. Sci. Statist. Comput., 11:5, 873-912.

6.19.82 `subroutine qrstep_zero_bd ( d, e, cs_left, sn_left, cs_right, sn_right, deflate, update_bd )`

Purpose

QRSTEP_ZERO_BD performs one implicit QR step with a zero shift on a n-by-n real (upper) bidiagonal matrix BD.

On entry, the arguments D and E contain, respectively, the main diagonal and superdiagonal of the bidiagonal matrix.

On output, the arguments D and E contain, respectively, the new main diagonal and superdiagonal of the updated (e.g. deflated) bidiagonal matrix, if DEFLATE is set to true or if the optional logical argument UPDATE_BD is used with the value true, otherwise they are not changed.

The two chains of n-1 planar rotations produced during the QR step with zero shift are saved in the arguments CS_LEFT, SN_LEFT, CS_RIGHT, SN_RIGHT.

Arguments

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix BD.

On exit, the new main diagonal of the bidiagonal matrix if DEFLATE=true or if UPDATE_BD=true. Otherwise, D is not changed.

E (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the n-1 superdiagonal elements of the bidiagonal matrix BD.

On exit, the new superdiagonal of the bidiagonal matrix if DEFLATE=true or if UPDATE_BD=true. Otherwise, E is not changed.

The size of E must be size(E) = size(D) - 1.

CS_LEFT (OUTPUT) real(stnd), dimension(:) On exit, the vector of the cosines coefficients of the chain of n-1 Givens rotations applied to the bidiagonal matrix BD on the left in the current QR step.

The size of CS_LEFT must be size(CS_LEFT) = size(E) = size(D) - 1.

SN_LEFT (OUTPUT) real(stnd), dimension(:) On exit, the vector of the sines coefficients of the chain of n-1 Givens rotations applied to the bidiagonal matrix BD on the left in the current QR step.

The size of SN_LEFT must be size(SN_LEFT) = size(E) = size(D) - 1.

CS_RIGHT (OUTPUT) real(stnd), dimension(:) On exit, the vector of the cosines coefficients of the chain of n-1 Givens rotations applied to the bidiagonal matrix BD on the right in the current QR step.

The size of CS_RIGHT must be size(CS_RIGHT) = size(E) = size(D) - 1.

SN_RIGHT (OUTPUT) real(stnd), dimension(:) On exit, the vector of the sines coefficients of the chain of n-1 Givens rotations applied to the bidiagonal matrix BD on the right in the current QR step.

The size of SN_RIGHT must be size(SN_RIGHT) = size(E) = size(D) - 1.

DEFLATE (OUTPUT) logical(lgl) On exit:

- DEFLATE = true : indicates that deflation occured at the end of the step.
- DEFLATE = false: indicates that the last superdiagonal element of the bidiagonal matrix is not small.

UPDATE_BD (INPUT, OPTIONAL) logical(lgl) On entry:

- UPDATE_BD = true : indicates that the bidiagonal matrix will be updated on exit.
- UPDATE_BD = false: indicates that the bidiagonal matrix will be updated on exit only if DEFLATE = true.

The default value for UPDATE_BD is false.

Further Details

This subroutine is adapted from the implicit zero-shift QR algorithm given in the reference (1).

For further details, see:

(1) **Demmel, J.W., and Kahan, W., 1990:** Accurate singular values of bidiagonal matrices. SIAM J. Sci. Statist. Comput., 11:5, 873-912.

6.19.83 subroutine upper_bd_deflate (d, e, singval, leftvec, rightvec, failure, &

Purpose

UPPER_BD_DEFLATE computes the left and right singular vectors of a real (upper) bidiagonal matrix BD corresponding to a specified singular value, using a deflation technique.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the bidiagonal matrix BD.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 superdiagonal elements of the bidiagonal matrix BD.

The size of E must be size(E) = size(D) - 1 = n - 1.

SINGVAL (INPUT) real(stnd) On entry, a singular value of the bidiagonal matrix. SINGVAL is assumed to be positive or zero.

LEFTVEC (OUTPUT) real(stnd), dimension(:) On exit, the computed left singular vector associated with the singular value SINGVAL.

The shape of LEFTVEC must verify: size(LEFTVEC) = size(D) = n .

RIGHTVEC (OUTPUT) real(stnd), dimension(:) On exit, the computed right singular vector associated with the singular value SINGVAL.

The shape of RIGHTVEC must verify: size(RIGHTVEC) = size(D) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the deflation procedure of the bidiagonal matrix.

MAX_QR_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_QR_STEPS controls the maximum number of QR sweeps for deflating the bidiagonal matrix for a given singular value.

The algorithm fails to converge if the total number of QR sweeps exceeds MAX_QR_STEPS.

The default is 4.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the bidiagonal matrix BD is scaled before computing the deflation parameters in order to avoid overflows.

The default is to scale the bidiagonal matrix.

Further Details

UPPER_BD_DEFLATE is a low-level subroutine used by BD_DEFLATE subroutines. Its use as a stand-alone method for computing singular vectors of a bidiagonal matrix is not recommended.

Note also that the sign of the singular vectors computed by this subroutine is arbitrary and not necessarily consistent between the left and right singular vectors. In order to compute consistent singular triplets, subroutine BD_DEFLATE must be used instead.

For further details, see:

(1) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

(2) **Mastronardi, M., Van Barel, M., Van Camp, E., and Vandebril, R., 2006:** On computing the eigenvectors of a class of structured matrices. Journal of Computational and Applied Mathematics, 189, 580-591.

(3) **Demmel, J.W., and Kahan, W., 1990:** Accurate singular values of bidiagonal matrices. SIAM J. Sci. Statist. Comput., 11:5, 873-912.

6.19.84 `subroutine upper_bd_deflate ( d, e, singval, leftvec, rightvec, failure, max_qr_steps, scaling )`

Purpose

UPPER_BD_DEFLATE computes the left and right singular vectors of a real (upper) bidiagonal matrix BD corresponding to specified singular values, using a deflation technique.

Arguments

D (INPUT) real(stnd), dimension(:) On entry, the diagonal elements of the bidiagonal matrix BD.

E (INPUT) real(stnd), dimension(:) On entry, the n-1 superdiagonal elements of the bidiagonal matrix BD.

The size of E must be size(E) = size(D) - 1 = n - 1.

SINGVAL (INPUT) real(stnd), dimension(:) On entry, selected singular values of the bidiagonal matrix. The singular values can be given in any order, but are assumed to be positive or zero.

The size of SINGVAL must verify: size(SINGVAL) <= size(D) = n .

LEFTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed left singular vectors. The left singular vector associated with the singular value SINGVAL(j) is stored in the j-th column of LEFTVEC.

The shape of LEFTVEC must verify:

- size(LEFTVEC, 1) = size(D) = n ,
- size(LEFTVEC, 2) = size(SINGVAL) .

RIGHTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed right singular vectors. The right singular vector associated with the singular value SINGVAL(j) is stored in the j-th column of RIGHTVEC.

The shape of RIGHTVEC must verify:

- size(RIGHTVEC, 1) = size(D) = n ,

- size(RIGHTVEC, 2) = size(SINGVAL) .

FAILURE (OUTPUT) logical(lgl), dimension(:) On exit:

- FAILURE(j) = FALSE : indicates successful exit for the jth singular triplet.
- FAILURE(j) = TRUE : indicates that the algorithm did not converge and full accuracy was not attained in the deflation procedure of the bidiagonal matrix for the jth singular triplet.

The size of FAILURE must verify: size(FAILURE) = size(SINGVAL) .

MAX_QR_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_QR_STEPS controls the maximum number of QR sweeps for deflating the bidiagonal matrix for a given singular value. The algorithm fails to converge if the total number of QR sweeps for all eigenvalues exceeds MAX_QR_STEPS * size(EIGVAL).

The default is 4.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if SCALING=true the bidiagonal matrix BD is scaled before computing the deflation parameters in order to avoid overflows.

The default is to scale the bidiagonal matrix.

Further Details

UPPER_BD_DEFLATE is a low-level subroutine used by BD_DEFLATE subroutines. Its use as a stand-alone method for computing singular vectors of a bidiagonal matrix is not recommended.

Note also that the sign of the singular vectors computed by this subroutine is arbitrary and not necessarily consistent between the left and right singular vectors. In order to compute consistent singular triplets, subroutine BD_DEFLATE must be used instead.

For further details, see:

(1) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

(2) **Mastronardi, M., Van Barel, M., Van Camp, E., and Vandebril, R., 2006:** On computing the eigenvectors of a class of structured matrices. Journal of Computational and Applied Mathematics, 189, 580-591.

(3) **Demmel, J.W., and Kahan, W., 1990:** Accurate singular values of bidiagonal matrices. SIAM J. Sci. Statist. Comput., 11:5, 873-912.

6.19.85 subroutine bd_deflate (upper, d, e, s, leftvec, rightvec, failure, max_qr_steps, ortho, scaling, inviter)

Purpose

BD_DEFLATE computes the left and right singular vectors of a real n-by-n bidiagonal matrix BD corresponding to specified singular values, using deflation techniques on the bidiagonal matrix BD.

Arguments

UPPER (INPUT) logical(lgl) On entry, if:

- UPPER = true : BD is upper bidiagonal ;
- UPPER = false : BD is lower bidiagonal.

D (INPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix BD.

E (INPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the bidiagonal matrix BD. E(1) is arbitrary.

The size of E must verify: size(E) = size(D) = n .

S (INPUT) real(stnd), dimension(:) On entry, selected singular values of the bidiagonal matrix BD. The singular values must be given in decreasing order and are assumed to be positive or zero.

The size of S must verify: size(S) <= size(D) = n .

LEFTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed left singular vectors. The left singular vector associated with the singular value S(j) is stored in the j-th column of LEFTVEC.

The shape of LEFTVEC must verify:

- size(LEFTVEC, 1) = size(D) = n ,
- size(LEFTVEC, 2) = size(S) .

RIGHTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed right singular vectors. The right singular vector associated with the singular value S(j) is stored in the j-th column of RIGHTVEC.

The shape of RIGHTVEC must verify:

- size(RIGHTVEC, 1) = size(D) = n ,
- size(RIGHTVEC, 2) = size(S) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the deflation procedure of the bidiagonal matrix.

MAX_QR_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_QR_STEPS controls the maximum number of QR sweeps for deflating the bidiagonal matrix for a given singular value. The algorithm fails to converge if the total number of QR sweeps for all singular values exceeds MAX_QR_STEPS * size(S).

The default is 4.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry, if:

- ORTHO=true, the bidiagonal matrix BD is deflated sequentially for all the specified singular values; this implies that the singular vectors of the bidiagonal matrix BD will be automatically orthogonal on exit.
- ORTHO=false, the bidiagonal matrix BD is deflated in parallel for the different clusters of singular values or isolated singular values; this implies that orthogonality of the singular vectors of bidiagonal matrix BD is preserved inside each cluster, but not automatically between clusters.

The default is ORTHO=false.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if:

- SCALING=true, the bidiagonal matrix BD is scaled before computing the deflation parameters in order to avoid overflows;
- SCALING=false, the bidiagonal matrix BD is not scaled.

The default is to scale the bidiagonal matrix.

INVITER (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INVITER=true, singular vectors corresponding to isolated singular values or singular vectors of bidiagonal matrices with zeros are computed by inverse iteration instead of deflation.
- INVITER=false, singular vectors corresponding to isolated singular values or singular vectors of bidiagonal matrices with zeros are computed by deflation.

The default is INVITER=true.

Further Details

The singular vectors are computed using deflation techniques applied to the bidiagonal matrix BD. The first deflation technique used in BD_DEFLATE combines an extension to bidiagonal matrices of Fernando's approach for computing eigenvectors of tridiagonal matrices with a deflation procedure by Givens rotations originally developed by Godunov and his collaborators (see references (1) and (2) for more details). If this deflation technique failed, QR iterations are used instead as described in (3) and (4).

Optionally, singular vectors corresponding to isolated singular values or singular vectors of bidiagonal matrices with zeros may be also computed by inverse iteration on the Golub-Kahan tridiagonal form of the bidiagonal matrix BD. This is the default since in these cases inverse iteration is safer and faster than the deflation algorithms.

The computation of the singular vectors is parallelized if OPENMP is used.

It is essential that singular values given on entry of BD_DEFLATE are computed to high relative accuracy. Subroutines BD_SINGVAL or BD_SINGVAL2 may be used for this purpose.

BD_DEFLATE may fail if some the singular values specified in parameter S are nearly identical or for clusters of small singular values.

For further details, on the deflation techniques used in BD_DEFLATE, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(2) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

(3) **Mastronardi, M., Van Barel, M., Van Camp, E., and Vandebril, R., 2006:** On computing the eigenvectors of a class of structured matrices. Journal of Computational and Applied Mathematics, 189, 580-591.

(4) **Demmel, J.W., and Kahan, W., 1990:** Accurate singular values of bidiagonal matrices. SIAM J. Sci. Statist. Comput., 11:5, 873-912.

6.19.86 subroutine bd_deflate2 (mat, tauq, taup, d, e, s, leftvec, rightvec, failure, max_qr_steps, ortho, scaling, inviter)

Purpose

BD_DEFLATE2 computes the left and right singular vectors of a full real m-by-n matrix MAT corresponding to specified singular values, using deflation techniques.

It is required that the original matrix MAT has been reduced to upper or lower bidiagonal form BD by an orthogonal transformation:

Q' * MAT * P = BD

where Q and P are orthogonal. This can be done with a call to BD_CMP with parameters TAUQ and TAUP, before calling BD_SINGVAL (or BD_SINGVAL2) for computing singular values and a call to BD_DEFLATE2 for computing selected singular vectors.

Alternatively and more simply, the user can call SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 subroutines, with parameters TAUQ and TAUP present, to compute the bidiagonal reduction and singular values in one step and, finally, call DEFLATE2 for computing all or selected singular vectors of MAT.

If m >= n, BD is upper bidiagonal and if m < n, BD is lower bidiagonal.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the original m-by-n matrix after reduction by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 subroutines. MAT must contains the vectors which define the elementary reflectors H(i) and G(i) whose products determine the matrices Q and P, as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 subroutines when the arguments TAUQ and TAUP are present in the call to these subroutines. MAT must be specified as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 and is not modified by the routine.

TAUQ (INPUT) real(stnd), dimension(:) TAUQ(i) must contain the scalar factor of the elementary reflector H(i) which determines Q, as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 in their array argument TAUQ.

The size of TAUQ must verify: size(TAUQ) = min(size(MAT,1) , size(MAT,2)) .

TAUP (INPUT) real(stnd), dimension(:) TAUP(i) must contain the scalar factor of the elementary reflector G(i), which determines P, as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 in their array argument TAUP.

The size of TAUP must verify: size(TAUP) = min(size(MAT,1) , size(MAT,2)) .

D (INPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix BD as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2.

The size of D must verify: size(D) = min(size(MAT,1) , size(MAT,2)) .

E (INPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the bidiagonal matrix BD as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2:

- if m >= n, E(i) = BD(i-1,i) for i = 2,3,. . . ,n;
- if m < n, E(i) = BD(i,i-1) for i = 2,3,. . . ,m.

E(1) is arbitrary.

The size of E must verify: size(E) = min(size(MAT,1) , size(MAT,2)) .

S (INPUT) real(stnd), dimension(:) On entry, selected singular values of the bidiagonal matrix BD. The singular values must be given in decreasing order and are assumed to be positive or zero.

The size of S must verify: size(S) <= min(size(MAT,1) , size(MAT,2)) .

LEFTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed left singular vectors. The left singular vector associated with the singular value S(j) is stored in the j-th column of LEFTVEC.

The shape of LEFTVEC must verify:

- size(LEFTVEC, 1) = size(MAT, 1) = m ,
- size(LEFTVEC, 2) = size(S) .

RIGHTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed right singular vectors. The right singular vector associated with the singular value S(j) is stored in the j-th column of RIGHTVEC.

The shape of RIGHTVEC must verify:

- size(RIGHTVEC, 1) = size(MAT, 2) = n ,
- size(RIGHTVEC, 2) = size(S) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the deflation procedure of the bidiagonal matrix BD.

MAX_QR_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_QR_STEPS controls the maximum number of QR sweeps for deflating the bidiagonal matrix BD for a given singular value. The algorithm fails to converge if the total number of QR sweeps for all singular values exceeds MAX_QR_STEPS * size(S).

The default is 4.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry:

- ORTHO=true, the bidiagonal matrix BD is deflated sequentially for all the specified singular values; this implies that the singular vectors of the bidiagonal matrix BD will be automatically orthogonal on exit.
- ORTHO=false, the bidiagonal matrix BD is deflated in parallel for the different clusters of singular values or isolated singular values; this implies that orthogonality of the singular vectors of bidiagonal matrix BD is preserved inside each cluster, but not automatically between clusters.

The default is ORTHO=false.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if:

- SCALING=true, the intermediate bidiagonal matrix BD is scaled before computing the deflation parameters in order to avoid overflows;
- SCALING=false, the intermediate bidiagonal matrix BD is not scaled.

The default is to scale the bidiagonal matrix.

INVITER (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INVITER=true, singular vectors corresponding to isolated singular values or singular vectors of bidiagonal matrices with zeros are computed by inverse iteration instead of deflation.
- INVITER=false, singular vectors corresponding to isolated singular values or singular vectors of bidiagonal matrices with zeros are computed by deflation.

The default is INVITER=true.

Further Details

The singular vectors are computed using deflation techniques applied implicitly to the associated tridiagonal forms BD' * BD and BD * BD' of the bidiagonal matrix BD. See description of the BD_DEFLATE subroutine for more details.

The computation of the singular vectors is parallelized if OPENMP is used.

It is essential that singular values given on entry of BD_DEFLATE2 are computed to high (relative) accuracy. Subroutines BD_SINGVAL or BD_SINGVAL2 may be used for this purpose.

BD_DEFLATE2 may fail if some the singular values specified in parameter S are nearly identical or for clusters of small singular values for some pathological matrices.

The deflation algorithms used in BD_DEFLATE2 are competitive with the inverse iteration procedure implemented in BD_INVITER2.

For further details on the deflation techniques or the blocked back-transformation algorithm used in BD_DEFLATE2, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(2) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

(3) **Mastronardi, M., Van Barel, M., Van Camp, E., and Vandebril, R., 2006:** On computing the eigenvectors of a class of structured matrices. Journal of Computational and Applied Mathematics, 189, 580-591.

(4) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.19.87 subroutine bd_deflate2 (mat, p, d, e, s, leftvec, rightvec, failure, max_qr_steps, ortho, scaling, inviter, tol_reortho)

Purpose

BD_DEFLATE2 computes the left and right singular vectors of a full real m-by-n matrix MAT with m>=n corresponding to specified singular values, using deflation techniques.

It is required that the original matrix MAT has been reduced to upper bidiagonal form BD by an orthogonal transformation:

Q' * MAT * P = BD

where Q and P are orthogonal. This can be done with a call to BD_CMP2 (or a call to BD_CMP followed by a call to ORTHO_GEN_BD), before calling BD_SVD, BD_SINGVAL (or BD_SINGVAL2) subroutines for computing singular values and BD_DEFLATE2 for computing selected singular vectors.

Alternatively and more simply, the user can call SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 subroutines, with parameter P present, to compute the bidiagonal reduction and singular values in one step and, finally, call DEFLATE2 for computing all or selected singular vectors of MAT.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the m-by-n orthogonal matrix Q after reduction by BD_CMP2, SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 (or by BD_CMP and ORTHO_GEN_BD). MAT is not modified by the routine.

The shape of MAT must verify: size(MAT, 1) >= size(MAT, 2) = n .

P (INPUT) real(stnd), dimension(:,:) On entry, the n-by-n orthogonal matrix P after reduction by BD_CMP2, SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 (or by BD_CMP and ORTHO_GEN_BD). If P has been computed by BD_CMP2, SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4, P can be stored in factored form or not. Both cases are handled by the subroutine. P is not modified by the routine.

The shape of P must verify: size(P, 1) = size(P, 2) = size(MAT, 2) = n .

D (INPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix BD as returned by BD_CMP, BD_CMP2, SELECT_SINGVAL_CMP3 or SELECT_SINGVAL4 subroutines.

The size of D must verify: size(D) = size(MAT, 2) = n .

E (INPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the bidiagonal matrix BD as returned by BD_CMP, BD_CMP2, SELECT_SINGVAL_CMP3 or SELECT_SINGVAL4 subroutines:

E(i) = BD(i-1,i) for i = 2,3,. . . ,n;

E(1) is arbitrary.

The size of E must verify: size(E) = size(MAT, 2) = n .

S (INPUT) real(stnd), dimension(:) On entry, selected singular values of the bidiagonal matrix BD. The singular values must be given in decreasing order and are assumed to be positive or zero.

The size of S must verify: size(S) <= size(MAT, 2) = n .

LEFTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed left singular vectors. The left singular vector associated with the singular value S(j) is stored in the j-th column of LEFTVEC.

The shape of LEFTVEC must verify:

- size(LEFTVEC, 1) = size(MAT, 1) = m ,
- size(LEFTVEC, 2) = size(S) .

RIGHTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed right singular vectors. The right singular vector associated with the singular value S(j) is stored in the j-th column of RIGHTVEC.

The shape of RIGHTVEC must verify:

- size(RIGHTVEC, 1) = size(MAT, 2) = n ,
- size(RIGHTVEC, 2) = size(S) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the deflation procedure of the bidiagonal matrix BD.

MAX_QR_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_QR_STEPS controls the maximum number of QR sweeps for deflating the bidiagonal matrix BD for a given singular value. The algorithm fails to converge if the total number of QR sweeps for all singular values exceeds MAX_QR_STEPS * size(S).

The default is 4.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry:

- ORTHO=true, the bidiagonal matrix BD is deflated sequentially for all the specified singular values; this implies that the singular vectors of the bidiagonal matrix BD will be automatically orthogonal on exit.
- ORTHO=false, the bidiagonal matrix BD is deflated in parallel for the different clusters of singular values or isolated singular values; this implies that orthogonality of the singular vectors of bidiagonal matrix BD is preserved inside each cluster, but not automatically between clusters.

The default is ORTHO=false.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if:

- SCALING=true, the intermediate bidiagonal matrix BD is scaled before computing the deflation parameters in order to avoid overflows;
- SCALING=false, the intermediate bidiagonal matrix BD is not scaled.

The default is to scale the bidiagonal matrix.

INVITER (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INVITER=true, singular vectors corresponding to isolated singular values or singular vectors of bidiagonal matrices with zeros are computed by inverse iteration instead of deflation.
- INVITER=false, singular vectors corresponding to isolated singular values or singular vectors of bidiagonal matrices with zeros are computed by deflation.

The default is INVITER=true.

TOL_REORTHO (INPUT, OPTIONAL) real(stnd) On entry, TOL_REORTHO is used to determine if the left singular vectors stored in LEFTVEC must be reortogonalized on exit in order to correct for the loss of orthogonality in the Ralha-Barlow one-sided bidiagonal reduction algorithm if MAT is nearly deficient. If one of the singular values, S(i), verifies the condition

S(i) <= TOL_REORTHO * S(1)

all the computed left singular vectors are reorthogonalized with a QR factorization. If S(1) is the largest singular value of MAT, this condition leads to the assertion that the rank of MAT is less than size(S) and is thus a nearly singular matrix if TOL_REORTHO is a small positive value of the order of the machine epsilon.

TOL_REORTHO must be greater or equal to zero and less than or equal to one. If TOL_REORTHO = 0. is used, the left singular vectors are reorthogonalized only if some singular values are almost zero. On the other hand, If TOL_REORTHO = 1. is used, the left singular vectors are always reorthogonalized. If TOL_REORTHO is specified as less than zero or greater than one, the default value is used.

The default value is the value of the module parameter tol_reortho_def if size(S) = n and tol_reortho_partial_def otherwise.

Further Details

The singular vectors are computed using deflation techniques applied implicitly to the associated tridiagonal forms BD' * BD and BD * BD' of the bidiagonal matrix BD. See description of the BD_DEFLATE subroutine for more details.

The computation of the singular vectors is parallelized if OPENMP is used.

It is essential that singular values given on entry of BD_DEFLATE2 are computed to high (relative) accuracy. Subroutines BD_SINGVAL or BD_SINGVAL2 may be used for this purpose.

BD_DEFLATE2 may fail if some the singular values specified in parameter S are nearly identical or for clusters of small singular values for some pathological matrices.

The deflation algorithms used in BD_DEFLATE2 are competitive with the inverse iteration procedure implemented in BD_INVITER2.

For further details on the deflation techniques or the blocked back-transformation algorithm used in BD_DEFLATE2, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(2) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

(3) **Mastronardi, M., Van Barel, M., Van Camp, E., and Vandebril, R., 2006:** On computing the eigenvectors of a class of structured matrices. Journal of Computational and Applied Mathematics, 189, 580-591.

(4) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.19.88 subroutine bd_deflate2 (mat, tauq, taup, rlmat, d, e, s, leftvec, rightvec, failure, tauo, max_qr_steps, ortho, scaling, inviter)

Purpose

BD_DEFLATE2 computes the left and right singular vectors of a full real m-by-n matrix MAT corresponding to specified singular values, using deflation techniques.

It is required that the original matrix MAT has been reduced to upper bidiagonal form BD by a two-step algorithm as performed by BD_CMP subroutine with parameters TAUQ, TAUP, RLMAT, and eventually TAUO, or more simply by SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 subroutines with the same arguments:

- If m >= n, a QR factorization of the real m-by-n matrix MAT is first computed

  MAT = O * R

where O is orthogonal and R is upper triangular. In a second step, the n-by-n upper triangular matrix R is reduced to upper bidiagonal form BD by an orthogonal transformation :

Q' * R * P = BD

where Q and P are orthogonal and BD is an upper bidiagonal matrix.

- If m < n, an LQ factorization of the real m-by-n matrix MAT is first computed

  MAT = L * O

where O is orthogonal and L is lower triangular. In a second step, the m-by-m lower triangular matrix L is reduced to upper bidiagonal form BD by an orthogonal transformation :

Q' * L * P = BD

where Q and P are orthogonal and BD is an upper bidiagonal matrix.

After this call to BD_CMP with parameters TAUQ, TAUP, RLMAT, and eventually TAUO, the user can call BD_SINGVAL or BD_SINGVAL2 subroutines for computing singular values of BD and, finally, BD_DEFLATE2 for computing all or selected singular vectors of MAT.

Alternatively and more simply, the user can call SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 subroutines, with parameters TAUQ, TAUP, RLMAT, and eventually TAUO, to perform the two-stage bidiagonal reduction and get singular values in one step and, finally, call BD_DEFLATE2 for computing all or selected singular vectors of MAT.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the original m-by-n matrix after a two-stage bidiagonal reduction by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 subroutines (e.g. when the matrix argument RLMAT is present in the call to these subroutines). The precise content of MAT is determined by the presence or absence of the optional argument TAUO:

- if the optional argument TAUO is absent, it is assumed that the first n columns (if m>=n) or the first m rows (if m<n) of the orthogonal matrix O is stored explicitly in the argument MAT on entry;
- if the optional argument TAUO is present, it is assumed that the orthogonal matrix O is stored in factored form, as a product of elementary reflectors, in the argument MAT on entry. In this case, MAT must contains the vectors which define the elementary reflectors W(i) whose products determine the matrix O, as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 subroutines when the matrix argument RLMAT is present in the call to these subroutines.

In both cases, MAT must be specified as returned by BD_CMP and is not modified by the routine.

TAUQ (INPUT) real(stnd), dimension(:) TAUQ(i) must contain the scalar factor of the elementary reflector H(i) which determines Q, as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 in their array argument TAUQ.

The size of TAUQ must verify: size(TAUQ) = min(size(MAT,1) , size(MAT,2)) .

TAUP (INPUT) real(stnd), dimension(:) TAUP(i) must contain the scalar factor of the elementary reflector G(i), which determines P, as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 in their array argument TAUP.

The size of TAUP must verify: size(TAUP) = min(size(MAT,1) , size(MAT,2)) .

RLMAT (INPUT) real(stnd), dimension(:,:) On entry, the elements on and below the diagonal, with the array TAUQ, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array TAUP, represent the orthogonal matrix P as a product of elementary reflectors. RLMAT must be specified as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2 and is not modified by the routine.

The shape of RLMAT must verify: size(RLMAT, 1) = size(RLMAT, 2) = min(size(MAT,1) , size(MAT,2)).

D (INPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the upper bidiagonal matrix BD as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2.

The size of D must verify: size(D) = min(size(MAT,1) , size(MAT,2)) .

E (INPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the upper bidiagonal matrix BD as returned by BD_CMP, SELECT_SINGVAL_CMP or SELECT_SINGVAL_CMP2:

E(i) = BD(i-1,i) for i = 2,3,. . . ,min(m,n);

E(1) is arbitrary.

The size of E must verify: size(E) = min(size(MAT,1) , size(MAT,2)) .

S (INPUT) real(stnd), dimension(:) On entry, selected singular values of the upper bidiagonal matrix BD. The singular values must be given in decreasing order and are assumed to be positive or zero.

The size of S must verify: size(S) <= min(size(MAT,1) , size(MAT,2)) .

LEFTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed left singular vectors. The left singular vector associated with the singular value S(j) is stored in the j-th column of LEFTVEC.

The shape of LEFTVEC must verify:

- size(LEFTVEC, 1) = size(MAT, 1) = m ,
- size(LEFTVEC, 2) = size(S) .

RIGHTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed right singular vectors. The right singular vector associated with the singular value S(j) is stored in the j-th column of RIGHTVEC.

The shape of RIGHTVEC must verify:

- size(RIGHTVEC, 1) = size(MAT, 2) = n ,
- size(RIGHTVEC, 2) = size(S) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit ;
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the deflation procedure of the bidiagonal matrix BD .

TAUO (INPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors W(i), which represent the orthogonal matrix O of the QR or LQ decomposition of MAT.

If the optional argument TAUO is present, it is assumed that the orthogonal matrix O is stored in factored form, as a product of elementary reflectors, in the argument MAT on entry.

If the optional argument TAUO is absent, it is assumed that the orthogonal matrix O is stored explicitly in the argument MAT on entry.

If the optional argument TAUO has been specified in the initial call to the BD_CMP subroutine, this optional argument TAUO must also be specified in the call to BD_DEFLATE2, otherwise the results will be incorrect.

See description of the argument MAT in the description of the BD_CMP subroutine, when the argument RLMAT is also present, for further details.

The size of TAUO must be min(size(MAT,1) , size(MAT,2)).

MAX_QR_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_QR_STEPS controls the maximum number of QR sweeps for deflating the bidiagonal matrix BD for a given singular value. The algorithm fails to converge if the total number of QR sweeps for all singular values exceeds MAX_QR_STEPS * size(S).

The default is 4.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry:

- ORTHO=true, the bidiagonal matrix BD is deflated sequentially for all the specified singular values; this implies that the singular vectors of the bidiagonal matrix BD will be automatically orthogonal on exit.
- ORTHO=false, the bidiagonal matrix BD is deflated in parallel for the different clusters of singular values or isolated singular values; this implies that orthogonality of the singular vectors of bidiagonal matrix BD is preserved inside each cluster, but not automatically between clusters.

The default is ORTHO=false.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if:

- SCALING=true, the intermediate bidiagonal matrix BD is scaled before computing the deflation parameters in order to avoid overflows;
- SCALING=false, the intermediate bidiagonal matrix BD is not scaled.

The default is to scale the bidiagonal matrix.

INVITER (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INVITER=true, singular vectors corresponding to isolated singular values or singular vectors of bidiagonal matrices with zeros are computed by inverse iteration instead of deflation.
- INVITER=false, singular vectors corresponding to isolated singular values or singular vectors of bidiagonal matrices with zeros are computed by deflation.

The default is INVITER=true.

Further Details

The singular vectors of BD are computed using deflation techniques applied implicitly to the associated tridiagonal forms BD' * BD and BD * BD' of the bidiagonal matrix BD. See description of the BD_DEFLATE subroutine for more details.

The singular vectors of MAT are finally computed by a blocked back-transformation algorithm.

The computation of the singular vectors of BD and the blocked back-transformation algorithm to find the singular vectors of MAT are parallelized if OPENMP is used.

It is essential that singular values given on entry of BD_DEFLATE2 are computed to high (relative) accuracy. Subroutines BD_SINGVAL or BD_SINGVAL2 may be used for this purpose.

BD_DEFLATE2 may fail if some the singular values specified in parameter S are nearly identical or for clusters of small singular values for some pathological matrices.

The deflation algorithms used in BD_DEFLATE2 are competitive with the inverse iteration procedure implemented in BD_INVITER2.

For further details on the deflation techniques or the blocked back-transformation algorithm used in BD_DEFLATE2, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(2) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

(3) **Mastronardi, M., Van Barel, M., Van Camp, E., and Vandebril, R., 2006:** On computing the eigenvectors of a class of structured matrices. Journal of Computational and Applied Mathematics, 189, 580-591.

(4) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.19.89 `subroutine bd_deflate2 ( mat, rmat, p, d, e, s, leftvec, rightvec, failure, tauo, max_qr_steps, ortho, scaling, inviter, tol_reortho )`

Purpose

BD_DEFLATE2 computes all or selected left and right singular vectors of a full real m-by-n matrix MAT with m>=n corresponding to specified singular values, using deflation techniques.

It is required that the original matrix MAT has been reduced to upper bidiagonal form BD by a two-step algorithm as performed by SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 subroutines with parameters P, RMAT, and eventually TAUO:

A QR factorization of the real m-by-n matrix MAT is first computed

MAT = O * R

where O is orthogonal and R is upper triangular. In a second step, the n-by-n upper triangular matrix R is reduced to upper bidiagonal form BD by an orthogonal transformation :

Q' * R * P = BD

where Q and P are orthogonal and BD is an upper bidiagonal matrix. Subroutines SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 computes O, Q, P, BD and also all or some of the singular values of R, which are also the singular values of MAT. Using this two-step factorization, BD_DEFLATE2 computes all or selected left and right singular vectors of R and apply to them a back-transformation algorithm to obtain the corresponding left and right singular vectors of MAT.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the original m-by-n matrix after the two-step bidiagonal reduction by SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 subroutines with arguments P, RMAT, and eventually TAUO. The precise content of MAT is determined by the presence or absence of the optional argument TAUO:

- if the optional argument TAUO is absent, it is assumed that the first n columns of the orthogonal matrix O is stored explicitly in the argument MAT on entry,
- if the optional argument TAUO is present, it is assumed that the orthogonal matrix O is stored in factored form, as a product of elementary reflectors, in the argument MAT on entry. In this case, MAT must contains the vectors which define the elementary reflectors W(i) whose products determine the matrix O, as returned by SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 subroutines when the matrix argument RLMAT is present in the call to these subroutines.

In both cases, MAT must be specified as returned by these subroutines and is not modified by the routine. See the description of the argument TAUO below for further details.

The shape of MAT must verify: size(MAT, 1) >= size(MAT, 2) = n .

RMAT (INPUT) real(stnd), dimension(:,:) On entry, the n-by-n orthogonal matrix Q after reduction by SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 subroutines. RMAT must be specified as returned by these subroutines and is not modified by the routine.

The shape of RMAT must verify: size(RMAT, 1) = size(RMAT, 2) = size(MAT, 2) = n.

P (INPUT) real(stnd), dimension(:,:) On entry, the n-by-n orthogonal matrix P after reduction by SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 subroutines. P can be stored in factored form or not. Both cases are handled by the subroutine and P is not modified by the routine.

The shape of P must verify: size(P, 1) = size(P, 2) = size(MAT, 2) = n .

D (INPUT) real(stnd), dimension(:) On entry, D contains the diagonal elements of the bidiagonal matrix BD as returned by SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4.

The size of D must verify: size(D) = size(MAT, 2) = n .

E (INPUT) real(stnd), dimension(:) On entry, E contains the off-diagonal elements of the bidiagonal matrix BD as returned by SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4:

E(i) = BD(i-1,i) for i = 2,3,. . . ,n;

E(1) is arbitrary.

The size of E must verify: size(E) = size(MAT, 2) = n .

S (INPUT) real(stnd), dimension(:) On entry, selected singular values of the bidiagonal matrix BD. The singular values must be given in decreasing order and are assumed to be positive or zero.

The size of S must verify: size(S) <= size(MAT, 2) .

LEFTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed left singular vectors. The left singular vector associated with the singular value S(j) is stored in the j-th column of LEFTVEC.

The shape of LEFTVEC must verify:

- size(LEFTVEC, 1) = size(MAT, 1) = m ,
- size(LEFTVEC, 2) = size(S) .

RIGHTVEC (OUTPUT) real(stnd), dimension(:,:) On exit, the computed right singular vectors. The right singular vector associated with the singular value S(j) is stored in the j-th column of RIGHTVEC.

The shape of RIGHTVEC must verify:

- size(RIGHTVEC, 1) = size(MAT, 2) = n ,
- size(RIGHTVEC, 2) = size(S) .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit,
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the deflation procedure of the bidiagonal matrix BD.

TAUO (INPUT, OPTIONAL) real(stnd), dimension(:) The scalar factors of the elementary reflectors W(i), which represent the orthogonal matrix O of the QR decomposition of MAT as returned by SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4.

If the optional argument TAUO is present, it is assumed that the orthogonal matrix O is stored in factored form, as a product of elementary reflectors, in the argument MAT on entry.

If the optional argument TAUO is absent, it is assumed that the orthogonal matrix O is stored explicitly in the argument MAT on entry.

If the optional argument TAUO has been specified in the initial call to the SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 subroutines, this optional argument TAUO must also be specified in the call to BD_DEFLATE2, otherwise the results will be incorrect.

See description of the argument MAT in the description of the SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 subroutines, when the argument RMAT is also present, for further details.

The size of TAUO must be size(MAT,2) = n .

MAX_QR_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_QR_STEPS controls the maximum number of QR sweeps for deflating the bidiagonal matrix BD for a given singular value. The algorithm fails to converge if the total number of QR sweeps for all singular values exceeds MAX_QR_STEPS * size(S).

The default is 4.

ORTHO (INPUT, OPTIONAL) logical(lgl) On entry:

- ORTHO=true, the bidiagonal matrix BD is deflated sequentially for all the specified singular values; this implies that the singular vectors of the bidiagonal matrix BD will be automatically orthogonal on exit;
- ORTHO=false, the bidiagonal matrix BD is deflated in parallel for the different clusters of singular values or isolated singular values; this implies that orthogonality of the singular vectors of bidiagonal matrix BD is preserved inside each cluster, but not automatically between clusters.

The default is ORTHO=false.

SCALING (INPUT, OPTIONAL) logical(lgl) On entry, if:

- SCALING=true, the intermediate bidiagonal matrix BD is scaled before computing the deflation parameters in order to avoid overflows;
- SCALING=false, the intermediate bidiagonal matrix BD is not scaled.

The default is to scale the bidiagonal matrix.

INVITER (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INVITER=true, singular vectors corresponding to isolated singular values or singular vectors of bidiagonal matrices with zeros are computed by inverse iteration instead of deflation.
- INVITER=false, singular vectors corresponding to isolated singular values or singular vectors of bidiagonal matrices with zeros are computed by deflation.

The default is INVITER=true.

TOL_REORTHO (INPUT, OPTIONAL) real(stnd) On entry, TOL_REORTHO is used to determine if the left singular vectors stored in LEFTVEC must be reortogonalized on exit in order to correct for the loss of orthogonality in the Ralha-Barlow one-sided bidiagonal reduction algorithm if MAT is nearly deficient. If one of the singular values, S(i), verifies the condition

S(i) <= TOL_REORTHO * S(1)

all the computed left singular vectors are reorthogonalized with a QR factorization. If S(1) is the largest singular value of MAT, this condition leads to the assertion that the rank of MAT is less than size(S) and is thus a nearly singular matrix if TOL_REORTHO is a small positive value of the order of the machine epsilon.

TOL_REORTHO must be greater or equal to zero and less than or equal to one. If TOL_REORTHO = 0. is used, the left singular vectors are reorthogonalized only if some singular values are almost zero. On the other hand, If TOL_REORTHO = 1. is used, the left singular vectors are always reorthogonalized. If TOL_REORTHO is specified as less than zero or greater than one, the default value is used.

The default value is the value of the module parameter tol_reortho_def if size(S) = n and tol_reortho_partial_def otherwise.

Further Details

The singular vectors are computed using deflation techniques applied implicitly to the associated tridiagonal forms BD' * BD and BD * BD' of the bidiagonal matrix BD. See description of the BD_DEFLATE subroutine for more details.

The computation of the singular vectors is parallelized if OPENMP is used.

It is essential that singular values given on entry of BD_DEFLATE2 are computed to high (relative) accuracy. Subroutines SELECT_SINGVAL_CMP3 or SELECT_SINGVAL_CMP4 may be used for this purpose.

BD_DEFLATE2 may fail if some the singular values specified in parameter S are nearly identical or for clusters of small singular values for some pathological matrices.

The deflation algorithms used in BD_DEFLATE2 are competitive with the inverse iteration procedure implemented in BD_INVITER2.

For further details on the deflation techniques or the blocked back-transformation algorithm used in BD_DEFLATE2, see:

(1) **Fernando, K.V., 1997:** On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. Siam J. Matrix Anal. Appl., Vol. 18, pp. 1013-1034.

(2) **Malyshev, A.N., 2000:** On deflation for symmetric tridiagonal matrices. Report 182 of the Department of Informatics, University of Bergen, Norway.

(3) **Mastronardi, M., Van Barel, M., Van Camp, E., and Vandebril, R., 2006:** On computing the eigenvectors of a class of structured matrices. Journal of Computational and Applied Mathematics, 189, 580-591.

(4) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore.

6.19.90 `subroutine svd_sort ( sort, d, u, v )`

Purpose

Given the singular values D and singular vectors U and V as output from BD_SVD, SVD_CMP or SVD_CMP3, this subroutine sorts the singular values into ascending or descending order, and, rearranges the columns of U and V correspondingly.

Arguments

SORT (INPUT) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

The singular vectors are rearranged accordingly.

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the singular values.

On exit, the singular values in ascending or decreasing order.

U (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the columns of U are the (left) singular vectors.

On exit, U contains the rearranged (left) singular vectors.

The shape of U must verify: size(U,2) = size(D) .

V (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the columns of V are the (right) singular vectors.

On exit, V contains the rearranged (right) singular vectors.

The shape of V must verify: size(V,2) = size(D) .

Further Details

The method is straight insertion.

6.19.91 subroutine svd_sort2 (sort, d, u, vt)

Purpose

Given the singular values D and singular vectors U and VT as output from BD_SVD2 or SVD_CMP2, this subroutine sorts the singular values into ascending or descending order, and, rearranges the columns of U and the rows of VT correspondingly.

Arguments

SORT (INPUT) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

The singular vectors are rearranged accordingly.

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the singular values.

On exit, the singular values in ascending or decreasing order.

U (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the columns of U are the (left) singular vectors.

On exit, U contains the rearranged (left) singular vectors.

The shape of U must verify: size(U,2) = size(D) .

VT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the rows of VT are the (right) singular vectors.

On exit, VT contains the rearranged (right) singular vectors.

The shape of VT must verify: size(VT,1) = size(D) .

Further Details

The method is straight insertion.

6.19.92 subroutine singvec_sort (sort, d, u)

Purpose

Given the singular values D and singular vectors U, stored columwise, as output from BD_SVD, SVD_CMP, BD_SVD2, SVD_CMP2 or SVD_CMP3, this subroutine sorts the singular values into ascending or descending order, and, rearranges the columns of U correspondingly.

Arguments

SORT (INPUT) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

The singular vectors are reordered accordingly.

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the singular values.

On exit, the singular values in ascending or decreasing order.

U (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the columns of U are the singular vectors.

On exit, U contains the reordered singular vectors.

The shape of U must verify: size(U,2) = size(D) = n .

Further Details

The method is straight insertion.

6.19.93 subroutine singval_sort (sort, d)

Purpose

Given the singular values D as output from BD_SVD, BD_SVD2, SVD_CMP, SVD_CMP2 or SVD_CMP3, this routine sorts the singular values into ascending or descending order.

Arguments

SORT (INPUT) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'.

D (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the singular values.

On exit, the singular values in ascending or decreasing order.

Further Details

The method is quick sort.

6.19.94 subroutine product_svd_cmp (a, b, s, failure, sort, maxiter, max_francis_steps, perfect_shift, bisect, dqds)

Purpose

This subroutine computes the singular value decomposition of the product of a m-by-n matrix A by the transpose of a p-by-n matrix B:

A * B' = U * SIGMA * V'

where A and B have more rows than columns (n<=min(m,p)), SIGMA is an n-by-n matrix which is zero except for its diagonal elements, U is an m-by-n orthogonal matrix, and V is an p-by-n orthogonal matrix. The diagonal elements of SIGMA are the singular values of A * B'; they are real and non-negative. The columns of U and V are the left and right singular vectors of A * B', respectively.

Arguments

A (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix A.

On exit, the m-by-n left-singular matrix U.

B (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the p-by-n matrix B.

On exit, the p-by-n right-singular matrix V.

S (OUTPUT) real(stnd), dimension(:) The singular values of A * B'.

The size of S must verify: size(S) = n .

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit
- FAILURE = true : indicates that the algorithm did not converge and that full accuracy was not attained in the SVD.

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd'. The singular vectors are rearranged accordingly.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm.

The bidiagonal SVD algorithm of an intermediate bidiagonal form of A * B' fails to converge if the number of QR sweeps exceeds MAXITER * n. Convergence usually occurs in about 2 * n QR sweeps.

The default is 10.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm.

MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of n and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

The size of S must match: size(S) = size(A, 2) = size(B, 2) .

6.19.95 function ginv (mat, tol, maxiter, max_francis_steps, perfect_shift, bisect, dqds)

Purpose

GINV returns the generalized inverse of a m-by-n real matrix, MAT. The generalized inverse of MAT is a n-by-m matrix.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

TOL (INPUT, OPTIONAL) real(stnd) On entry:

- If TOL is less than or equal to zero or is absent, the function computes the generalized inverse of MAT.
- If TOL is greater than zero, the subroutine computes the generalized inverse of a matrix close to MAT, but having condition number in the 2-norm less than 1/TOL.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm.

The bidiagonal SVD phase of an intermediate bidiagonal form B of MAT fails to converge if the number of QR sweeps exceeds MAXITER * min(m,n). Convergence usually occurs in about 2 * min(m,n) QR sweeps.

The default is 10.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm.

MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of min(m,n) and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed generalized inverse at the expense of a slightly slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

If MAT is the null matrix or the SVD algorithm used to compute the generalized inverse of MAT did not converge and full accuracy was not attained in the bidiagonal SVD of an intermediate bidiagonal form of MAT, function GINV returns a n-by-m matrix filled with NAN() function.

The computation of the generalized inverse is parallelized if OPENMP is used.

For further details, on the generalized inverse of a rectangular matrix and the algorithm to compute it, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore, Maryland.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

6.19.96 `subroutine comp_ginv ( mat, failure, matginv, tol, singvalues, krank, mul_size, maxiter, max_francis_steps, perfect_shift, bisect, dqds )`

Purpose

COMP_GINV computes the generalized inverse of a m-by-n real matrix, MAT.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT is destroyed.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that MAT is the null matrix or that the SVD algorithm which is used to compute the generalized inverse of MAT did not converge and that full accuracy was not attained in the bidiagonal SVD of an intermediate bidiagonal form B of MAT.

MATGINV (OUTPUT) real(stnd), dimension(:,:) On exit, MATGINV contains the generalized inverse of MAT or the generalized inverse of a matrix close to MAT.

The shape of MATGINV must verify:

- size(MATGINV,1) = size(MAT,2) = n ,
- size(MATGINV,2) = size(MAT,1) = m .

TOL (INPUT, OPTIONAL) real(stnd) On entry, if:

- TOL is less than or equal to zero or is absent, the subroutine computes the generalized inverse of MAT.
- TOL is greater than zero, the subroutine computes the generalized inverse of a matrix close to MAT, but having condition number in the 2-norm less than 1/TOL.

SINGVALUES (OUTPUT, OPTIONAL) real(stnd), dimension(:) The singular values of MAT in decreasing order. The condition number of MAT in the 2-norm is

SINGVALUES(1)/SINGVALUES(min(m,n)).

The size of SINGVALUES must verify : size(SINGVALUES) = min(m,n) .

KRANK (OUTPUT, OPTIONAL) integer(i4b) On exit, the effective rank of MAT, i.e., the number of singular values which are greater than TOL * SINGVALUES(1).

MUL_SIZE (INPUT, OPTIONAL) integer(i4b) Internal parameter. MUL_SIZE must verify: 1 <= MUL_SIZE <= max(m,n), otherwise a default value is used. MUL_SIZE can be increased or decreased to improve the performance of the algorithm.

The default value is 32.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm.

The bidiagonal SVD algorithm of an intermediate bidiagonal form B of MAT fails to converge if the number of QR sweeps exceeds MAXITER * min(m,n). Convergence usually occurs in about 2 * min(m,n) QR sweeps.

The default is 10.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm.

MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of min(m,n) and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed generalized inverse at the expense of a slightly slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

If all the elements of MAT are equal to zero, subroutine COMP_GINV returns a n-by-m matrix filled with NAN() function in argument MATGINV and the logical argument FAILURE is set to .true. .

The computation of the generalized inverse is parallelized if OPENMP is used.

For further details, on the generalized inverse of a rectangular matrix and the algorithm to compute it, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore, Maryland.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

6.19.97 subroutine comp_ginv (mat, failure, tol, singvalues, krank, mul_size, maxiter, max_francis_steps, perfect_shift, bisect, dqds)

Purpose

COMP_GINV computes the generalized inverse of a m-by-n real matrix, MAT.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the m-by-n matrix MAT.

On exit, MAT contains the transpose of the generalized inverse of MAT or of the generalized inverse of a matrix close to MAT.

FAILURE (OUTPUT) logical(lgl) On exit:

- FAILURE = false : indicates successful exit.
- FAILURE = true : indicates that MAT is the null matrix or that the SVD algorithm which is used to compute the generalized inverse of MAT did not converge and that full accuracy was not attained in the bidiagonal SVD of an intermediate bidiagonal form B of MAT.

TOL (INPUT, OPTIONAL) real(stnd) On entry, if:

- TOL is less than or equal to zero or is absent, the subroutine computes the generalized inverse of MAT.
- TOL is greater than zero, the subroutine computes the generalized inverse of a matrix close to MAT, but having condition number in the 2-norm less than 1/TOL.

SINGVALUES (OUTPUT, OPTIONAL) real(stnd), dimension(:) The singular values of MAT in decreasing order. The condition number of MAT in the 2-norm is

SINGVALUES(1)/SINGVALUES(min(m,n)).

The size of SINGVALUES must verify: size(SINGVALUES) = min(m,n) .

KRANK (OUTPUT, OPTIONAL) integer(i4b) On exit, the effective rank of MAT, i.e., the number of singular values which are greater than TOL * SINGVALUES(1).

MUL_SIZE (INPUT, OPTIONAL) integer(i4b) Internal parameter. MUL_SIZE must verify: 1 <= MUL_SIZE <= max(m,n), otherwise a default value is used. MUL_SIZE can be increased or decreased to improve the performance of the algorithm.

The default value is 32.

MAXITER (INPUT, OPTIONAL) integer(i4b) MAXITER controls the maximum number of QR sweeps in the bidiagonal SVD phase of the SVD algorithm.

The bidiagonal SVD algorithm of an intermediate bidiagonal form B of MAT fails to converge if the number of QR sweeps exceeds MAXITER * min(m,n). Convergence usually occurs in about 2 * min(m,n) QR sweeps.

The default is 10.

MAX_FRANCIS_STEPS (INPUT, OPTIONAL) integer(i4b) MAX_FRANCIS_STEPS controls the maximum number of Francis sets (e.g. QR sweeps) of Givens rotations which must be saved before applying them with a wavefront algorithm to accumulate the singular vectors in the bidiagonal SVD algorithm.

MAX_FRANCIS_STEPS is a strictly positive integer, otherwise the default value is used.

The default is the minimum of n and the integer parameter MAX_FRANCIS_STEPS_SVD specified in the module Select_Parameters.

PERFECT_SHIFT (INPUT, OPTIONAL) logical(lgl) PERFECT_SHIFT determines if a perfect shift strategy is used in the implicit QR algorithm in order to minimize the number of QR sweeps in the bidiagonal SVD algorithm.

The default is true.

BISECT (INPUT, OPTIONAL) logical(lgl) BISECT determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If BISECT is set to true, singular values are computed with a more accurate bisection algorithm delivering improved accuracy in the final computed generalized inverse at the expense of a slightly slower execution time.

If BISECT is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

DQDS (INPUT, OPTIONAL) logical(lgl) DQDS determines how the singular values are computed if a perfect shift strategy is used in the bidiagonal SVD algorithm (e.g., if PERFECT_SHIFT is equal to TRUE). This argument has no effect if PERFECT_SHIFT is equal to false.

If DQDS is set to true, singular values are computed with a more accurate dqds algorithm delivering improved accuracy in the final computed SVD decomposition at the expense of a slightly slower execution time.

If DQDS is set to false, singular values are computed with the fast Pal-Walker-Kahan algorithm.

If both optional arguments BISECT and DQDS are specified with the value true, the bisection algorithm is used.

The default is false.

Further Details

If all the elements of MAT are equal to zero, subroutine COMP_GINV returns a m-by-n matrix filled with NAN() function in argument MAT and the logical argument FAILURE is set to .true. .

The computation of the generalized inverse is parallelized if OPENMP is used.

For further details, on the generalized inverse of a rectangular matrix and the algorithm to compute it, see:

(1) **Golub, G.H., and Van Loan, C.F., 1996:** Matrix Computations. 3rd ed. The Johns Hopkins University Press, Baltimore, Maryland.

(2) **Lawson, C.L., and Hanson, R.J., 1974:** Solving least square problems. Prentice-Hall.

6.19.98 `subroutine gen_bd_mat ( type, d, e, failure, known_singval, from_tridiag, singval, sort, val1, val2, l0, glu0 )`

Purpose

GEN_BD_MAT generates different types of bidiagonal matrices with known singular values or specific numerical properties such as clustered singular values for testing purposes of singular value decomposition bidiagonal solvers.

Optionally, the singular values of the selected bidiagonal matrix can be computed analytically, if possible, or by a bisection algorithm with high absolute and relative accuracies.

Arguments

TYPE (INPUT) integer(i4b) Select the type of bidiagonal matrix BD to be generated by the subroutine.

If TYPE is between 1 and 56, the subroutine generates a specific bidiagonal matrix as described in the comments inside the code of the subroutine. For other values of TYPE, all diagonal and off-diagonal elements of the bidiagonal matrix are generated from an uniform random numbers distribution between 0 and 1.

For TYPE between 1 and 17, the singular values of the bidiagonal matrix are known analytically. For other values of TYPE, the singular values may be estimated by a bisection algorithm with high accuracy. In all cases, the singular values may be output in the optional parameter SINGVAL.

For TYPE between 1 and 11 or 52 and 56, the bidiagonal matrix BD is computed as the Cholesky factor of symmetric positive-definite tridiagonal matrices.

D (OUTPUT) real(stnd), dimension(:) On exit, D contains the diagonal elements of the bidiagonal matrix BD.

The size of D must verify: size(D)>=2 .

E (OUTPUT) real(stnd), dimension(:) On exit, E(2:) contains the off-diagonal elements of the bidiagonal matrix BD. E(1) is arbitrary, but is set to zero.

The size of E must verify: size(E) = size(D) .

FAILURE (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FAILURE = false : indicates that the singular values of BD are known analytically or have been computed with high accuracy;
- FAILURE = true : indicates that the singular values of BD are not known analytically and have not been computed with maximum accuracy with the bisection algorithm.

KNOWN_SINGVAL (OUTPUT, OPTIONAL) logical(lgl) On exit:

- KNOWN_SINGVAL = true : indicates that the singular values of BD are known analytically for the selected TYPE.
- KNOWN_SINGVAL = false : indicates that the eigenvalues of BD are not known analytically for the selected TYPE.

FROM_TRIDIAG (OUTPUT, OPTIONAL) logical(lgl) On exit:

- FROM_TRIDIAG = true : indicates that the bidiagonal matrix BD has been computed as the Cholesky factor of a positive-definite tridiagonal matrix for the selected TYPE.
- FROM_TRIDIAG = false : indicates that the bidiagonal matrix BD has not been computed as the Cholesky factor of a positive-definite tridiagonal matrix for the selected TYPE.

SINGVAL (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the singular values of BD computed analytically or estimated to high accuracy with a bisection algorithm.

The size of SINGVAL must verify: size(SINGVAL) = size(D) .

SORT (INPUT, OPTIONAL) character Sort the singular values into ascending order if SORT = 'A' or 'a', or in descending order if SORT = 'D' or 'd', if the optional argument SINGVAL is present. For other values of SORT nothing is done and SINGVAL(:) may not be sorted.

VAL1 (INPUT, OPTIONAL) real(stnd) On entry, specifies the parameter d0 for parametrized bidiagonal matrices (e.g. TYPE= 2-8, 10, 15, 32-35).

If this parameter is changed for TYPE between 2 and 8, care must be taken to insure that the initial symmetric tridiagonal matrix, which is used to derive the bidiagonal matrix BD, is positive-definite. If this is not the case, the subroutine will issue an error message and stop the program.

Also, if this parameter is changed for TYPE between 32 and 35, which correspond to graded (or reversely graded) matrices with an arithmetic or geometric progression, care must be taken to insure that some elements of the arithmetic or geometric progression will not underflow or overflow as no checks are done in the subroutine for such errors.

The default for VAL1 is:

- 2. for TYPE between 2 and 7;
- 3. for TYPE equal to 8;
- 1. for TYPE equal to 10;
- 1. for TYPE equal to 15;
- 1. for TYPE between 32 and 35.

VAL2 (INPUT, OPTIONAL) real(stnd) On entry, specifies the parameter e0 for parametrized bidiagonal matrices (e.g. TYPE= 2-8, 10, 32-35).

If this parameter is changed for TYPE between 2 and 8, care must be taken to insure that the initial symmetric tridiagonal matrix, which is used to derive the bidiagonal matrix BD, is positive-definite. If this is not the case, the subroutine will issue an error message and stop the program.

Also, if this parameter is changed for TYPE between 32 and 35, which correspond to graded (or reversely graded) matrices with an arithmetic or geometric progression, care must be taken to insure that some elements of the arithmetic or geometric progression will not underflow or overflow as no checks are done in the subroutine for such errors.

The default for VAL2 is:

- 1. for TYPE between 2 and 7;
- 2. for TYPE equal to 8;
- 2. for TYPE equal to 10;
- 2. for TYPE between 32 and 35.

L0 (INPUT, OPTIONAL) integer(i4b) On entry, specify the radius of the initial matrix for parametrized form of glued bidiagonal matrices (e.g. for TYPE equal to 44, 46, 48, 53, 55).

L0 must be greater than 0 and preferably less or equal to size(D)/2 . The default is 5.

GLU0 (INPUT, OPTIONAL) real(stnd) On entry, specify the glue parameter for parametrized form of glued bidiagonal matrices (e.g. for TYPE equal to 44, 46, 48, 53, 55).

The default is sqrt(epsilon(GLU0)).

Further Details

This subroutine tries to take care of imprecisions in intrinsic subroutines (e.g. like the cos function in the gfortran compiler) when computing singular values by analytic formulae.

For further details on the bidiagonal matrices used for testing in GEN_BD_MAT subroutine, see:

(1) **Gladwell, G.M.L., Jones, T.H., Willms N.B., 2014:** A test matrix for an inverse eigenvalue problem. Journal of Applied Mathematics, 14, 6 pages, Article ID 515082, DOI 10.1155/2014/515082.

(2) **Clement, P.A., 1959:** A class of triple-diagonal matrices for test purposes. SIAM Review, 1(1):50-52, DOI 10.1137/1001006.

(3) **Gregory, R.T., Karney, D.L., 1969:** A collection of matrices for testing computational algorithms. New York: Wiley. Reprinted with corrections by Robert E. Krieger, Huntington, New York, 1978.

(4) **Higham, N.J., 1991: Algorithm 694:** A collection of test matrices in MATLAB. ACM Transactions on Mathematical Software 17(3):289-305 DOI 10.1145/114697.116805.

(5) **Godunov, S.K., Antonov, A.G., Kirillyuk, O.P., and Kostin, V.I., 1993:** Guaranteed Accuracy in numerical linear algebra. Kluwer Academic Publishers.

(6) **Parlett, B.N., and Vomel, C., 2005:** How the MRRR algorithm can fail on tight eigenvalue clusters. Lapack Working Note 163.

(7) **Nakatsukasa, Y., Aishima, K., and Yamazaki, I., 2012:** dqds with agressive early deflation. SIAM J. Matrix Anal. Appl., 33(1): 22-51.

(8) **Fernando, K.V., and Parlett, B.N., 1994:** Accurate singular values and differenial qd algorithms. Numer. Math., 67: 191-229.

6.20 Module_Select_Parameters

Copyright 2025 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

THIS MODULE PROVIDES A CONVENIENT WAY OF SELECTING :

- THE PRECISION (KIND TYPES) REQUIRED FOR A COMPUTATION.
- THE SIZE (KIND TYPES) OF INTEGER OR LOGICAL VARIABLES.
- THE DEFAULT PRINTING UNIT.
- THE DIFFERENT BLOCK SIZES FOR LINEAR ALGEBRA SUBROUTINES.
- THE PARAMETERS FOR OpenMP COMPILATION.
- THE PARAMETERS FOR CROSSOVER FROM SERIAL TO PARALLEL ALGORITHMS.
- THE PARAMETERS FOR THE STATPACK TESTING PROGRAMS.
- THE LOCATION OF THE URANDOM DEVICE ON YOUR SYSTEM IF IT EXISTS.

IN ORDER TO CHANGE THE DEFAULT VALUES AND MAKE YOUR OWN CHOICE FOR THESE PARAMETERS, YOU MUST EDIT THE FILE Module_Select_Parameters.F90 AND FOLLOW THE INSTRUCTIONS IN THIS FILE.

LATEST REVISION : 03/09/2025

6.21 Module_Sort_Procedures

Copyright 2024 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING SORTING UTILITIES.

LATEST REVISION : 12/03/2024

6.21.1 `subroutine tri_insert ( list )`

Purpose

Sort the integer array LIST into ascending numerical order, by straight insertion. LIST is replaced on output by its sorted rearrangement.

Arguments

LIST (INPUT/OUTPUT) integer(i4b), dimension(:) The integer vector to sort.

6.21.2 `subroutine tri_insert ( list, order )`

Purpose

Sort the integer array LIST into ascending numerical order, by straight insertion. LIST is replaced on output by its sorted rearrangement.

ORDER is an associated integer array which gives the positions of the elements in the original order.

Arguments

LIST (INPUT/OUTPUT) integer(i4b), dimension(:) The integer vector to sort.

ORDER (OUTPUT) integer(i4b), dimension(:) Array which gives the positions of the elements in the original order.

Further Details

The size of LIST and ORDER must match.

6.21.3 subroutine tri_insert (list)

Purpose

Sort the real array LIST into ascending numerical order, by straight insertion. LIST is replaced on output by its sorted rearrangement.

Arguments

LIST (INPUT/OUTPUT) real(stnd), dimension(:) The real vector to sort.

6.21.4 subroutine tri_insert (list, order)

Purpose

Sort the real array LIST into ascending numerical order, by straight insertion. LIST is replaced on output by its sorted rearrangement.

ORDER is an associated integer array which gives the positions of the elements in the original order.

Arguments

LIST (INPUT/OUTPUT) real(stnd), dimension(:) The real vector to sort.

ORDER (OUTPUT) integer(i4b), dimension(:) Array which gives the positions of the elements in the original order.

Further Details

The size of LIST and ORDER must match.

6.21.5 subroutine quick_sort (list, ascending)

Purpose

Sort an integer array LIST into ascending or descending order using the QuickSort algorithm. LIST is replaced on output by its sorted rearrangement.

Arguments

LIST (INPUT/OUTPUT) integer(i4b), dimension(:) The integer vector to sort.

ASCENDING (INPUT, OPTIONAL) logical(lgl) Sort the array list into ascending order if ASCENDING = true, or in descending order if ASCENDING = false. The default is true.

Further Details

Quick sort routine adapted (and modified to reverse order) from:

(1) **Brainerd, W.S., Goldberg, C.H., and Adams, J.C., 1990:** Programmer's Guide to Fortran 90. McGraw-Hill, ISBN 0-07-000248-7, pages 149-150.

6.21.6 `subroutine quick_sort ( list, ascending )`

Purpose

Sort a real array LIST into ascending or descending order using the QuickSort algorithm. LIST is replaced on output by its sorted rearrangement.

Arguments

LIST (INPUT/OUTPUT) real(stnd), dimension(:) The real vector to sort.

ASCENDING (INPUT, OPTIONAL) logical(lgl) Sort the array list into ascending order if ASCENDING = true, or in descending order if ASCENDING = false. The default is true.

Further Details

Quick sort routine adapted (and modified to reverse order) from:

(1) **Brainerd, W.S., Goldberg, C.H., and Adams, J.C., 1990:** Programmer's Guide to Fortran 90. McGraw-Hill, ISBN 0-07-000248-7, pages 149-150.

6.21.7 `subroutine quick_sort ( list, order, ascending )`

Purpose

Sort an integer array LIST into ascending or descending order using the QuickSort algorithm. LIST is replaced on output by its sorted rearrangement.

ORDER is an associated integer array which gives the positions of the elements in the original order.

Arguments

LIST (INPUT/OUTPUT) integer(i4b), dimension(:) The integer vector to sort.

ORDER (OUTPUT) integer(i4b), dimension(:) Array which gives the positions of the elements in the original order.

ASCENDING (INPUT, OPTIONAL) logical(lgl) Sort the array list into ascending order if ASCENDING = true, or in descending order if ASCENDING = false. The default is true.

Further Details

Quick sort routine adapted from reference (1), modified to include an associated integer array, which gives the positions of the elements in the original order, and also modified to reverse order.

The sizes of LIST and ORDER must match.

For further details, see:

(1) **Brainerd, W.S., Goldberg, C.H., and Adams, J.C., 1990:** Programmer's Guide to Fortran 90. McGraw-Hill, ISBN 0-07-000248-7, pages 149-150.

6.21.8 `subroutine quick_sort ( list, order, ascending )`

Purpose

Sort a real array LIST into ascending or descending order using the QuickSort algorithm. LIST is replaced on output by its sorted rearrangement.

ORDER is an associated integer array which gives the positions of the elements in the original order.

Arguments

LIST (INPUT/OUTPUT) real(stnd), dimension(:) The real vector to sort.

ORDER (OUTPUT) integer(i4b), dimension(:) Array which gives the positions of the elements in the original order.

ASCENDING (INPUT, OPTIONAL) logical(lgl) Sort the array list into ascending order if ASCENDING = true, or in descending order if ASCENDING = false. The default is true.

Further Details

Quick sort routine adapted from reference (1), modified to include an associated integer array, which gives the positions of the elements in the original order, and also modified to reverse order.

The sizes of LIST and ORDER must match.

For further details, see:

(1) **Brainerd, W.S., Goldberg, C.H., and Adams, J.C., 1990:** Programmer's Guide to Fortran 90. McGraw-Hill, ISBN 0-07-000248-7, pages 149-150.

6.21.9 `subroutine do_index ( list, index )`

Purpose

This subroutine indexes an integer array LIST, i.e., outputs the array INDEX of length N such that LIST(INDEX(j)) is in ascending order for j=1, 2, . . . , N. The input quantity LIST is not changed.

Arguments

LIST (INPUT) integer(i4b), dimension(:) The integer vector to index.

INDEX (OUTPUT) integer(i4b), dimension(:) The index array.

Further Details

The sizes of LIST and INDEX must match.

6.21.10 subroutine do_index (list, index)

Purpose

This subroutine indexes a real array LIST, i.e., outputs the array INDEX of length N such that LIST(INDEX(j)) is in ascending order for j=1, 2, . . . , N. The input quantity LIST is not changed.

Arguments

LIST (INPUT) real(stnd), dimension(:) The real vector to index.

INDEX (OUTPUT) integer(i4b), dimension(:) The index array.

Further Details

The sizes of LIST and INDEX must match.

6.21.11 function rank (index)

Purpose

Given INDEX as output from the routine DO_INDEX, this routine returns a same-size array RANK, the corresponding table of ranks.

Arguments

INDEX (INPUT) integer(i4b), dimension(:) The index.

Further Details

This function is adapted from Numerical Recipes.

6.21.12 subroutine reorder (index, slave, ascending)

Purpose

Given INDEX as output from the routine DO_INDEX, this routine makes the corresponding rearrangement of the same-size integer array SLAVE. The rearrangement is performed by means of the integer array INDEX.

Arguments

INDEX (INPUT) integer(i4b), dimension(:) The index vector.

SLAVE (INPUT/OUTPUT) integer(i4b), dimension(:) Integer vector to rearrange according to INDEX.

ASCENDING (INPUT, OPTIONAL) logical(lgl) Rearrange SLAVE according to ascending order if ASCENDING = true, or to descending order if ASCENDING = false. The default is true.

Further Details

The size of SLAVE and INDEX must match.

6.21.13 `subroutine reorder ( index, slave, ascending )`

Purpose

Given INDEX as output from the routine DO_INDEX, this routine makes the corresponding rearrangement of the same-size real array SLAVE. The rearrangement is performed by means of the integer array INDEX.

Arguments

INDEX (INPUT) integer(i4b), dimension(:) The index vector.

SLAVE (INPUT/OUTPUT) real(stnd), dimension(:) Real vector to rearrange according to INDEX.

ASCENDING (INPUT, OPTIONAL) logical(lgl) Rearrange SLAVE according to ascending order if ASCENDING = true, or to descending order if ASCENDING = false. The default is true.

Further Details

The sizes of SLAVE and INDEX must match.

6.21.14 `subroutine reorder ( index, slave, ascending )`

Purpose

Given INDEX as output from the routine DO_INDEX, this routine makes the corresponding rearrangement of the same-size complex array SLAVE. The rearrangement is performed by means of the integer array INDEX.

Arguments

INDEX (INPUT) integer(i4b), dimension(:) The index vector.

SLAVE (INPUT/OUTPUT) complex(stnd), dimension(:) Complex vector to rearrange according to INDEX.

ASCENDING (INPUT, OPTIONAL) logical(lgl) Rearrange SLAVE according to ascending order if ASCENDING = true, or to descending order if ASCENDING = false. The default is true.

Further Details

The sizes of SLAVE and INDEX must match.

6.21.15 subroutine reorder (index, slave, ascending)

Purpose

Given INDEX as output from the routine DO_INDEX, this routine makes the corresponding rearrangement of the columns of the integer matrix SLAVE. The rearrangement is performed by means of the integer array INDEX.

Arguments

INDEX (INPUT) integer(i4b), dimension(:) The index vector.

SLAVE (INPUT/OUTPUT) integer(i4b), dimension(:,:) Integer matrix to rearrange according to INDEX.

ASCENDING (INPUT, OPTIONAL) logical(lgl) Rearrange the columns of SLAVE according to ascending order if ASCENDING = true, or to descending order if ASCENDING = false.

The default is true.

Further Details

The size of INDEX must match the number of columns of SLAVE. The rearrangement is done in place.

6.21.16 subroutine reorder (index, slave, ascending)

Purpose

Given INDEX as output from the routine DO_INDEX, this routine makes the corresponding rearrangement of the columns of the real matrix SLAVE. The rearrangement is performed by means of the integer array INDEX.

Arguments

INDEX (INPUT) integer(i4b), dimension(:) The index vector.

SLAVE (INPUT/OUTPUT) real(stnd), dimension(:,:) Real matrix to rearrange according to INDEX.

ASCENDING (INPUT, OPTIONAL) logical(lgl) Rearrange the columns of SLAVE according to ascending order if ASCENDING = true, or to descending order if ASCENDING = false.

The default is true.

Further Details

The size of INDEX must match the number of columns of SLAVE. The rearrangement is done in place.

6.21.17 subroutine reorder (index, slave, ascending)

Purpose

Given INDEX as output from the routine DO_INDEX, this routine makes the corresponding rearrangement of the columns of the complex matrix SLAVE. The rearrangement is performed by means of the integer array INDEX.

Arguments

INDEX (INPUT) integer(i4b), dimension(:) The index vector.

SLAVE (INPUT/OUTPUT) complex(stnd), dimension(:,:) Complex matrix to rearrange according to INDEX.

ASCENDING (INPUT, OPTIONAL) logical(lgl) Rearrange the columns of SLAVE according to ascending order if ASCENDING = true, or to descending order if ASCENDING = false.

The default is true.

Further Details

The size of INDEX must match the number of columns of SLAVE. The rearrangement is done in place.

6.22 Module_Stat_Procedures

Copyright 2021 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING SUBROUTINES AND FUNCTIONS FOR UNIVARIATE STATISTICAL COMPUTATIONS

LATEST REVISION : 20/08/2021

6.22.1 subroutine comp_unistat (x, first, last, xstat, xnobs, nobias)

Purpose

COMP_UNISTAT computes estimates of univariate statistics from a data vector.

Arguments

X (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X) observations from the vector of data for which basic univariate statistics are desired. If all the data are available at once, X can be the full data vector.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subvector is the first subvector of the data vector.
- FIRST = false the current subvector is not the first subvector of the data vector.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current subvector is the last subvector of the data vector.
- LAST = false the current subvector is not the last subvector of the data vector.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(7) On entry, after the first call to COMP_UNISTAT (e.g. when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_UNISTAT. XSTAT should not be changed between calls to COMP_UNISTAT.

On exit, when LAST=true, XSTAT contains the following statistics :

- XSTAT(1) contains the mean value of the data vector.
- XSTAT(2) contains the variance of the data vector.
- XSTAT(3) contains the standard deviation of the data vector.
- XSTAT(4) contains the coefficient of skewness of the data vector.
- XSTAT(5) contains the coefficient of kurtosis of the data vector.
- XSTAT(6) contains the minimum of the data vector.
- XSTAT(7) contains the maximum of the data vector.

The size of XSTAT must verify: size(XSTAT) = 7.

XNOBS (OUTPUT, OPTIONAL) integer(i4b) On exit, XNOBS contains the number of observations in the data vector. XNOBS needs to be specified only on the last call to COMP_UNISTAT (e.g. when LAST=true).

NOBIAS (INPUT, OPTIONAL) logical(lgl) On entry, when LAST=true and NOBIAS=true, unbiased estimates of skewness and kurtosis are computed. If NOBIAS=false or is absent, biased estimates are computed.

NOBIAS needs to be specified only on the last call to COMP_UNISTAT (e.g. when LAST=true).

Further Details

The subroutine computes the basic statistics with only one pass through the data.

If fewer than four valid observations were present, the pertinent statistics are set to Nan code.

6.22.2 subroutine comp_unistat (x, first, last, xstat, dimvar, xnobs, nobias)

Purpose

COMP_UNISTAT computes estimates of univariate statistics from a data matrix.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which basic univariate statistics are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrix is the first submatrix of the data matrix.
- FIRST = false the current submatrix is not the first submatrix of the data matrix.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrix is the last submatrix of the data matrix.
- LAST = false the current submatrix is not the last submatrix of the data matrix.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,7) On entry, after the first call to COMP_UNISTAT (e.g. when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_UNISTAT. XSTAT should not be changed between calls to COMP_UNISTAT.

On exit, when LAST=true, each column of XSTAT contains the following statistics on all variables:

- XSTAT(:,1) contains the mean values.
- XSTAT(:,2) contains the variances.
- XSTAT(:,3) contains the standard deviations.
- XSTAT(:,4) contains the coefficients of skewness.
- XSTAT(:,5) contains the coefficients of kurtosis.
- XSTAT(:,6) contains the minima.
- XSTAT(:,7) contains the maxima.

The shape of XSTAT must verify:

- size(XSTAT,1) = size(X,DIMVAR) ;
- size(XSTAT,2) = 7 .

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

XNOBS (OUTPUT, OPTIONAL) integer(i4b) On exit, XNOBS contains the number of observations in the data matrix. XNOBS needs to be specified only on the last call to COMP_UNISTAT (e.g. when LAST=true).

NOBIAS (INPUT, OPTIONAL) logical(lgl) On entry, when LAST=true and NOBIAS=true, unbiased estimates of skewness and kurtosis are computed. If NOBIAS=false or is absent, biased estimates are computed.

NOBIAS needs to be specified only on the last call to COMP_UNISTAT (e.g. when LAST=true).

Further Details

The subroutine computes the basic statistics with only one pass through the data.

If fewer than four valid observations were present, the pertinent statistics are set to Nan code.

6.22.3 subroutine comp_unistat (x, first, last, xstat, xnobs, nobias)

Purpose

COMP_UNISTAT computes estimates of univariate statistics from a tridimensional data array.

Arguments

X (INPUT) real(stnd), dimension(:,:,:) On entry, input subarray containing size(X,3) observations on size(X,1) by size(X,2) variables from the array of data for which basic univariate statistics are desired. If all the data are available at once, X can be the full data array.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subarray is the first subarray of the data array.
- FIRST = false the current subarray is not the first subarray of the data array.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current subarray is the last subarray of the data array.
- LAST = false the current subarray is not the last subarray of the data array.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,:,7) On entry, after the first call to COMP_UNISTAT (e.g. when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_UNISTAT. XSTAT should not be changed between calls to COMP_UNISTAT.

On exit, when LAST=true, each matrix of XSTAT contains the following statistics on all variables:

- XSTAT(:,:,1) contains the mean values.
- XSTAT(:,:,2) contains the variances.
- XSTAT(:,:,3) contains the standard deviations.
- XSTAT(:,:,4) contains the coefficients of skewness.
- XSTAT(:,:,5) contains the coefficients of kurtosis.
- XSTAT(:,:,6) contains the minima.

- XSTAT(:,:,7) contains the maxima.

The shape of XSTAT must verify:

- size(XSTAT,1) = size(X,1)
- size(XSTAT,2) = size(X,2)
- size(XSTAT,3) = 7 .

XNOBS (OUTPUT, OPTIONAL) integer(i4b) On exit, XNOBS contains the number of observations in the data array. XNOBS needs to be specified only on the last call to COMP_UNISTAT (e.g. when LAST=true).

NOBIAS (INPUT, OPTIONAL) logical(lgl) On entry, when LAST=true and NOBIAS=true, unbiased estimates of skewness and kurtosis are computed. If NOBIAS=false or is absent, biased estimates are computed.

NOBIAS needs to be specified only on the last call to COMP_UNISTAT (e.g. when LAST=true).

Further Details

The subroutine computes the basic statistics with only one pass through the data.

If fewer than four valid observations were present, the pertinent statistics are set to Nan code.

6.22.4 subroutine comp_unistat (x, first, last, xstat, xnobs, nobias)

Purpose

COMP_UNISTAT computes estimates of univariate statistics from a fourdimensional data array.

Arguments

X (INPUT) real(stnd), dimension(:,:,:,:) On entry, input subarray containing size(X,4) observations on size(X,1) by size(X,2) by size(X,3) variables from the array of data for which basic univariate statistics are desired. If all the data are available at once, X can be the full data array.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subarray is the first subarray of the data array.
- FIRST = false the current subarray is not the first subarray of the data array.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current subarray is the last subarray of the data array.
- LAST = false the current subarray is not the last subarray of the data array.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,:,:,7) On entry, after the first call to COMP_UNISTAT (e.g. when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_UNISTAT. XSTAT should not be changed between calls to COMP_UNISTAT.

On exit, when LAST=true, each matrix of XSTAT contains the following statistics on all variables:

- XSTAT(:,:,:,1) contains the mean values.

- XSTAT(:,:,:,2) contains the variances.
- XSTAT(:,:,:,3) contains the standard deviations.
- XSTAT(:,:,:,4) contains the coefficients of skewness.
- XSTAT(:,:,:,5) contains the coefficients of kurtosis.
- XSTAT(:,:,:,6) contains the minima.
- XSTAT(:,:,:,7) contains the maxima.

The shape of XSTAT must verify:

- size(XSTAT,1) = size(X,1),
- size(XSTAT,2) = size(X,2)
- size(XSTAT,3) = size(X,3)
- size(XSTAT,4) = 7 .

XNOBS (OUTPUT, OPTIONAL) integer(i4b) On exit, XNOBS contains the number of observations in the data array. XNOBS needs to be specified only on the last call to COMP_UNISTAT (e.g. when LAST=true).

NOBIAS (INPUT, OPTIONAL) logical(lgl) On entry, when LAST=true and NOBIAS=true, unbiased estimates of skewness and kurtosis are computed. If NOBIAS=false or is absent, biased estimates are computed.

NOBIAS needs to be specified only on the last call to COMP_UNISTAT (e.g. when LAST=true).

Further Details

The subroutine computes the basic statistics with only one pass through the data.

If fewer than four valid observations were present, the pertinent statistics are set to Nan code.

6.22.5 `subroutine comp_unistat ( x, first, last, xstat, xmiss, xnobs, nobias )`

Purpose

COMP_UNISTAT_MISS computes estimates of univariate statistics from a data vector possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X) observations from the vector of data for which basic univariate statistics are desired. If all the data are available at once, X can be the full data vector.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subvector is the first subvector of the data vector.
- FIRST = false the current subvector is not the first subvector of the data vector.

LAST (INPUT) logical(lgl) On entry, if:

- **LAST = true the current subvector is the last subvector** of the data vector.

- **LAST = false the current subvector is not the last subvector** of the data vector.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(7) On entry, after the first call to COMP_UNISTAT_MISS (e.g. when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_UNISTAT_MISS. XSTAT should not be changed between calls to COMP_UNISTAT_MISS.

On exit, when LAST=true, XSTAT contains the following statistics :

- XSTAT(1) contains the mean value of the data vector.
- XSTAT(2) contains the variance of the data vector.
- XSTAT(3) contains the standard deviation of the data vector.
- XSTAT(4) contains the coefficient of skewness of the data vector.
- XSTAT(5) contains the coefficient of kurtosis of the data vector.
- XSTAT(6) contains the minimum of the data vector.
- XSTAT(7) contains the maximum of the data vector.

The size of XSTAT must verify: size(XSTAT) = 7.

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

XNOBS (OUTPUT, OPTIONAL) integer(i4b) On exit, XNOBS contains the number of non-missing observations in the data vector. XNOBS needs to be specified only on the last call to COMP_UNISTAT_MISS (e.g. when LAST=true).

NOBIAS (INPUT, OPTIONAL) logical(lgl) On entry, when LAST=true and NOBIAS=true, unbiased estimates of skewness and kurtosis are computed. If NOBIAS=false or is absent, biased estimates are computed.

NOBIAS needs to be specified only on the last call to COMP_UNISTAT_MISS (e.g. when LAST=true).

Further Details

The subroutine computes the basic statistics with only one pass through the data.

If fewer than four valid observations were present, the pertinent statistics are set to XMISS.

6.22.6 subroutine comp_unistat (x, first, last, xstat, xmiss, dimvar, xnobs, nobias)

Purpose

COMP_UNISTAT_MISS computes estimates of univariate statistics from a data matrix possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which basic univariate statistics are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrix is the first submatrix of the data matrix.
- FIRST = false the current submatrix is not the first submatrix of the data matrix.

LAST (INPUT) logical(lgl) On entry, if:

- **LAST = true the current submatrix is the last submatrix** of the data matrix.
- **LAST = false the current submatrix is not the last submatrix** of the data matrix.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,7) On entry, after the first call to COMP_UNISTAT_MISS (e.g. when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_UNISTAT_MISS. XSTAT should not be changed between calls to COMP_UNISTAT_MISS.

On exit, when LAST=true, each column of XSTAT contains the following statistics on all variables:

- XSTAT(:,1) contains the mean values.
- XSTAT(:,2) contains the variances.
- XSTAT(:,3) contains the standard deviations.
- XSTAT(:,4) contains the coefficients of skewness.
- XSTAT(:,5) contains the coefficients of kurtosis.
- XSTAT(:,6) contains the minima.
- XSTAT(:,7) contains the maxima.

The shape of XSTAT must verify:

- size(XSTAT,1) = size(X,DIMVAR)
- size(XSTAT,2) = 7 .

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

XNOBS (OUTPUT, OPTIONAL) integer(i4b), dimension(:) On exit, XNOBS contains the number of non-missing observations on all variables. XNOBS needs to be specified only on the last call to COMP_UNISTAT_MISS (e.g. when LAST=true).

The size of XNOBS must verify: size(XNOBS) = size(X,DIMVAR).

NOBIAS (INPUT, OPTIONAL) logical(lgl) On entry, when LAST=true and NOBIAS=true, unbiased estimates of skewness and kurtosis are computed. If NOBIAS=false or is absent, biased estimates are computed.

NOBIAS needs to be specified only on the last call to COMP_UNISTAT (e.g. when LAST=true).

Further Details

The subroutine computes the basic statistics with only one pass through the data.

If fewer than four valid observations were present for some variables, the pertinent statistics are set to XMISS.

6.22.7 subroutine comp_unistat (x, first, last, xstat, xmiss, xnobs, nobias)

Purpose

COMP_UNISTAT_MISS computes estimates of univariate statistics from a tridimensional data array possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:,:) On entry, input subarray containing size(X,3) observations on size(X,1) by size(X,2) variables from the array of data for which basic univariate statistics are desired. If all the data are available at once, X can be the full tridimensional data array.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subarray is the first subarray of the data array.
- FIRST = false the current subarray is not the first subarray of the data array.

LAST (INPUT) logical(lgl) On entry, if:

- **LAST = true the current subarray is the last subarray** of the data array.
- **LAST = false the current subarray is not the last subarray** of the data array.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,:,7) On entry, after the first call to COMP_UNISTAT_MISS (e.g. when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_UNISTAT_MISS. XSTAT should not be changed between calls to COMP_UNISTAT_MISS.

On exit, when LAST=true, each matrix of XSTAT contains the following statistics on all variables:

- XSTAT(:,:,1) contains the mean values.
- XSTAT(:,:,2) contains the variances.
- XSTAT(:,:,3) contains the standard deviations.
- XSTAT(:,:,4) contains the coefficients of skewness.
- XSTAT(:,:,5) contains the coefficients of kurtosis.
- XSTAT(:,:,6) contains the minima.
- XSTAT(:,:,7) contains the maxima.

The shape of XSTAT must verify:

- size(XSTAT,1) = size(X,1)
- size(XSTAT,2) = size(X,2)
- size(XSTAT,3) = 7 .

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

XNOBS (OUTPUT, OPTIONAL) integer(i4b), dimension(:,:) On exit, XNOBS contains the numbers of non-missing observations on all variables. XNOBS needs to be specified only on the last call to COMP_UNISTAT_MISS (e.g. when LAST=true).

The shape of XNOBS must verify:

- size(XNOBS,1) = size(X,1)
- size(XNOBS,2) = size(X,2).

NOBIAS (INPUT, OPTIONAL) logical(lgl) On entry, when LAST=true and NOBIAS=true, unbiased estimates of skewness and kurtosis are computed. If NOBIAS=false or is absent, biased estimates are computed.

NOBIAS needs to be specified only on the last call to COMP_UNISTAT_MISS (e.g. when LAST=true).

Further Details

The subroutine computes the basic statistics with only one pass through the data.

If fewer than four valid observations were present for some variables, the pertinent statistics are set to XMISS.

6.22.8 subroutine comp_unistat (x, first, last, xstat, xmiss, xnobs, nobias)

Purpose

COMP_UNISTAT_MISS computes estimates of univariate statistics from a fourdimensional data array possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:,:,:) On entry, input subarray containing size(X,4) observations on size(X,1) by size(X,2) by size(X,3) variables from the array of data for which basic univariate statistics are desired. If all the data are available at once, X can be the full tridimensional data array.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subarray is the first subarray of the data array.
- FIRST = false the current subarray is not the first subarray of the data array.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current subarray is the last subarray of the data array.
- LAST = false the current subarray is not the last subarray of the data array.

XSTAT (INPUT/OUTPUT) real(stnd), dimension(:,:,:,7) On entry, after the first call to COMP_UNISTAT_MISS (e.g. when FIRST=true), XSTAT is used as workspace to accumulate quantities on previous calls to COMP_UNISTAT_MISS. XSTAT should not be changed between calls to COMP_UNISTAT_MISS.

On exit, when LAST=true, each matrix of XSTAT contains the following statistics on all variables:

- XSTAT(:,:,:,1) contains the mean values.
- XSTAT(:,:,:,2) contains the variances.
- XSTAT(:,:,:,3) contains the standard deviations.
- XSTAT(:,:,:,4) contains the coefficients of skewness.
- XSTAT(:,:,:,5) contains the coefficients of kurtosis.
- XSTAT(:,:,:,6) contains the minima.
- XSTAT(:,:,:,7) contains the maxima.

The shape of XSTAT must verify:

- size(XSTAT,1) = size(X,1)
- size(XSTAT,2) = size(X,2)
- size(XSTAT,3) = size(X,3)
- size(XSTAT,4) = 7 .

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

XNOBS (OUTPUT, OPTIONAL) integer(i4b), dimension(:,:,:) On exit, XNOBS contains the numbers of non-missing observations on all variables. XNOBS needs to be specified only on the last call to COMP_UNISTAT_MISS (e.g. when LAST=true).

The shape of XNOBS must verify:

- size(XNOBS,1) = size(X,1)
- size(XNOBS,2) = size(X,2)
- size(XNOBS,3) = size(X,3).

NOBIAS (INPUT, OPTIONAL) logical(lgl) On entry, when LAST=true and NOBIAS=true, unbiased estimates of skewness and kurtosis are computed. If NOBIAS=false or is absent, biased estimates are computed.

NOBIAS needs to be specified only on the last call to COMP_UNISTAT_MISS (e.g. when LAST=true).

Further Details

The subroutine computes the basic statistics with only one pass through the data.

If fewer than four valid observations were present for some variables, the pertinent statistics are set to XMISS.

6.22.9 `subroutine comp_mvs ( x, first, last, xmean, xvar, xstd, xnobs )`

Purpose

COMP_MVS computes estimates of mean, variance and standard-deviation from a data vector.

Arguments

X (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X) observations from the vector of data for which basic univariate statistics are desired. If all the data are available at once, X can be the full data vector.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subvector is the first subvector of the data vector.
- FIRST = false the current subvector is not the first subvector of the data vector.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current subvector is the last subvector of the data vector.
- LAST = false the current subvector is not the last subvector of the data vector.

XMEAN (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_MVS (e.g. when FIRST=true), XMEAN is used as workspace to accumulate quantity on previous calls to COMP_MVS. XMEAN should not be changed between calls to COMP_MVS.

On exit, when LAST=true, XMEAN contains the mean value of the data vector.

XVAR (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_MVS (e.g. when FIRST=true), XVAR is used as workspace to accumulate quantity on previous calls to COMP_MVS. XVAR should not be changed between calls to COMP_MVS.

On exit, when LAST=true, XVAR contains the variance of the data vector.

XSTD (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_MVS (e.g. when FIRST=true), XSTD is used as workspace to accumulate quantity on previous calls to COMP_MVS. XSTD should not be changed between calls to COMP_MVS.

On exit, when LAST=true, XSTD contains the standard deviation of the data vector.

XNOBS (OUTPUT, OPTIONAL) integer(i4b) On exit, XNOBS contains the number of observations in the data vector. XNOBS needs to be specified only on the last call to COMP_MVS (e.g. when LAST=true).

Further Details

The subroutine computes the basic statistics with only one pass through the data.

If fewer than one valid observation is present, the statistics are set to Nan code.

6.22.10 `subroutine comp_mvs ( x, first, last, xmean, xvar, xstd, dimvar, xnobs )`

Purpose

COMP_MVS computes estimates of means, variances and standard-deviations from a data matrix.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which basic univariate statistics

are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrix is the first submatrix of the data matrix.
- FIRST = false the current submatrix is not the first submatrix of the data matrix.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrix is the last submatrix of the data matrix.
- LAST = false the current submatrix is not the last submatrix of the data matrix.

XMEAN (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_MVS (e.g. when FIRST=true), XMEAN is used as workspace to accumulate quantities on previous calls to COMP_MVS. XMEAN should not be changed between calls to COMP_MVS.

On exit, when LAST=true, XMEAN contains the mean values.

The size of XMEAN must verify: size(XMEAN) = size(X,DIMVAR).

XVAR (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_MVS (e.g. when FIRST=true), XVAR is used as workspace to accumulate quantities on previous calls to COMP_MVS. XVAR should not be changed between calls to COMP_MVS.

On exit, when LAST=true, XVAR contains the variances.

The size of XVAR must verify: size(XVAR) = size(X,DIMVAR).

XSTD (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_MVS (e.g. when FIRST=true), XSTD is used as workspace to accumulate quantities on previous calls to COMP_MVS. XSTD should not be changed between calls to COMP_MVS.

On exit, when LAST=true, XSTD contains the standard deviations.

The size of XSTD must verify: size(XSTD) = size(X,DIMVAR).

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

XNOBS (OUTPUT, OPTIONAL) integer(i4b) On exit, XNOBS contains the number of observations in the data matrix. XNOBS needs to be specified only on the last call to COMP_MVS (e.g. when LAST=true).

Further Details

The subroutine computes the basic statistics with only one pass through the data.

If fewer than one valid observation is present, the statistics are set to Nan code.

6.22.11 subroutine comp_mvs (x, first, last, xmean, xvar, xstd, xnobs)

Purpose

COMP_MVS computes estimates of means, variances and standard-deviations from a tridimensional data array.

Arguments

X (INPUT) real(stnd), dimension(:,:,:) On entry, input subarray containing size(X,3) observations on size(X,1) by size(X,2) variables from the array of data for which basic univariate statistics are desired. If all the data are available at once, X can be the full data array.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subarray is the first subarray of the data array.
- FIRST = false the current subarray is not the first subarray of the data array.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current subarray is the last subarray of the data array.
- LAST = false the current subarray is not the last subarray of the data array.

XMEAN (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_MVS (e.g. when FIRST=true), XMEAN is used as workspace to accumulate quantities on previous calls to COMP_MVS. XMEAN should not be changed between calls to COMP_MVS.

On exit, when LAST=true, XMEAN contains the mean values.

The shape of XMEAN must verify:

- size(XMEAN,1) = size(X,1)
- size(XMEAN,2) = size(X,2).

XVAR (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_MVS (e.g. when FIRST=true), XVAR is used as workspace to accumulate quantities on previous calls to COMP_MVS. XVAR should not be changed between calls to COMP_MVS.

On exit, when LAST=true, XVAR contains the variances.

The shape of XVAR must verify:

- size(XVAR,1) = size(X,1)
- size(XVAR,2) = size(X,2).

XSTD (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_MVS (e.g. when FIRST=true), XSTD is used as workspace to accumulate quantities on previous calls to COMP_MVS. XSTD should not be changed between calls to COMP_MVS.

On exit, when LAST=true, XSTD contains the standard deviations.

The shape of XSTD must verify:

- size(XSTD,1) = size(X,1)
- size(XSTD,2) = size(X,2).

XNOBS (OUTPUT, OPTIONAL) integer(i4b) On exit, XNOBS contains the number of observations in the data array. XNOBS needs to be specified only on the last call to COMP_MVS (e.g. when LAST=true).

Further Details

The subroutine computes the basic statistics with only one pass through the data.

If fewer than one valid observation is present, the statistics are set to Nan code.

6.22.12 subroutine comp_mvs (x, first, last, xmean, xvar, xstd, xnobs)

Purpose

COMP_MVS computes estimates of means, variances and standard-deviations from a fourdimensional data array.

Arguments

X (INPUT) real(stnd), dimension(:,:,:,:) On entry, input subarray containing size(X,4) observations on size(X,1) by size(X,2) by size(X,3) variables from the array of data for which basic univariate statistics are desired. If all the data are available at once, X can be the full data array.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subarray is the first subarray of the data array.
- FIRST = false the current subarray is not the first subarray of the data array.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current subarray is the last subarray of the data array.
- LAST = false the current subarray is not the last subarray of the data array.

XMEAN (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, after the first call to COMP_MVS (e.g. when FIRST=true), XMEAN is used as workspace to accumulate quantities on previous calls to COMP_MVS. XMEAN should not be changed between calls to COMP_MVS.

On exit, when LAST=true, XMEAN contains the mean values.

The shape of XMEAN must verify:

- size(XMEAN,1) = size(X,1)
- size(XMEAN,2) = size(X,2)
- size(XMEAN,3) = size(X,3).

XVAR (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, after the first call to COMP_MVS (e.g. when FIRST=true), XVAR is used as workspace to accumulate quantities on previous calls to COMP_MVS. XVAR should not be changed between calls to COMP_MVS.

On exit, when LAST=true, XVAR contains the variances.

The shape of XVAR must verify:

- size(XVAR,1) = size(X,1)
- size(XVAR,2) = size(X,2)
- size(XVAR,3) = size(X,3).

XSTD (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, after the first call to COMP_MVS (e.g. when FIRST=true), XSTD is used as workspace to accumulate quantities on previous calls to COMP_MVS. XSTD should not be changed between calls to COMP_MVS.

On exit, when LAST=true, XSTD contains the standard deviations.

The shape of XSTD must verify:

- size(XSTD,1) = size(X,1)
- size(XSTD,2) = size(X,2)
- size(XSTD,3) = size(X,3).

XNOBS (OUTPUT, OPTIONAL) integer(i4b) On exit, XNOBS contains the number of observations in the data array. XNOBS needs to be specified only on the last call to COMP_MVS (e.g. when LAST=true).

Further Details

The subroutine computes the basic statistics with only one pass through the data.

If fewer than one valid observation is present, the statistics are set to Nan code.

6.22.13 subroutine comp_mvs (x, first, last, xmean, xvar, xstd, xmiss, xnobs)

Purpose

COMP_MVS_MISS computes estimates of mean, variance and standard-deviation from a data vector possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X) observations from the vector of data for which basic univariate statistics are desired. If all the data are available at once, X can be the full data vector.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subvector is the first subvector of the data vector.
- FIRST = false the current subvector is not the first subvector of the data vector.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current subvector is the last subvector of the data vector.
- LAST = false the current subvector is not the last subvector of the data vector.

XMEAN (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_MVS_MISS (e.g. when FIRST=true), XMEAN is used as workspace to accumulate quantity on previous calls to COMP_MVS_MISS. XMEAN should not be changed between calls to COMP_MVS_MISS.

On exit, when LAST=true, XMEAN contains the mean value of the data vector.

XVAR (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_MVS_MISS (e.g. when FIRST=true), XVAR is used as workspace to accumulate quantity on previous calls to COMP_MVS_MISS. XVAR should not be changed between calls to COMP_MVS_MISS.

On exit, when LAST=true, XVAR contains the variance of the data vector.

XSTD (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_MVS_MISS (e.g. when FIRST=true), XSTD is used as workspace to accumulate quantity on previous calls to COMP_MVS_MISS. XSTD should not be changed between calls to COMP_MVS_MISS.

On exit, when LAST=true, XSTD contains the standard deviation of the data vector.

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

XNOBS (OUTPUT, OPTIONAL) integer(i4b) On exit, XNOBS contains the number of non-missing observations in the data vector. XNOBS needs to be specified only on the last call to COMP_MVS_MISS (e.g. when LAST=true).

Further Details

The subroutine computes the basic statistics with only one pass through the data.

If fewer than one valid (non-missing) observation is present, the pertinent statistics are set to XMISS.

6.22.14 subroutine comp_mvs (x, first, last, xmean, xvar, xstd, xmiss, dimvar, xnobs)

Purpose

COMP_MVS_MISS computes estimates of means, variances and standard-deviations from a data matrix possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which basic univariate statistics are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrix is the first submatrix of the data matrix.
- FIRST = false the current submatrix is not the first submatrix of the data matrix.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrix is the last submatrix of the data matrix.
- LAST = false the current submatrix is not the last submatrix of the data matrix.

XMEAN (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_MVS_MISS (e.g. when FIRST=true), XMEAN is used as workspace to accumulate quantities on previous calls to COMP_MVS_MISS. XMEAN should not be changed between calls to COMP_MVS_MISS.

On exit, when LAST=true, XMEAN contains the mean values.

The size of XMEAN must verify: size(XMEAN) = size(X,DIMVAR).

XVAR (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_MVS_MISS (e.g. when FIRST=true), XVAR is used as workspace to accumulate quantities on previous calls to COMP_MVS_MISS. XVAR should not be changed between calls to COMP_MVS_MISS.

On exit, when LAST=true, XVAR contains the variances.

The size of XVAR must verify: size(XVAR) = size(X,DIMVAR).

XSTD (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_MVS_MISS (e.g. when FIRST=true), XSTD is used as workspace to accumulate quantities on previous calls to COMP_MVS_MISS. XSTD should not be changed between calls to COMP_MVS_MISS.

On exit, when LAST=true, XSTD contains the standard deviations.

The size of XSTD must verify: size(XSTD) = size(X,DIMVAR).

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

XNOBS (OUTPUT, OPTIONAL) integer(i4b), dimension(:) On exit, XNOBS contains the numbers of non-missing observations on all variables. XNOBS needs to be specified only on the last call to COMP_MVS_MISS (e.g. when LAST=true).

The size of XNOBS must verify: size(XNOBS) = size(X,DIMVAR).

Further Details

The subroutine computes the basic statistics with only one pass through the data.

If fewer than one valid (non-missing) observation is present for some variables, the pertinent statistics are set to XMISS.

6.22.15 `subroutine comp_mvs ( x, first, last, xmean, xvar, xstd, xmiss, xnobs )`

Purpose

COMP_MVS_MISS computes estimates of means, variances and standard-deviations from a tridimensional data array possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:,:) On entry, input subarray containing size(X,3) observations on size(X,1) by size(X,2) variables from the array of data for which basic univariate statistics are desired. If all the data are available at once, X can be the full data array.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subarray is the first subarray of the data array.

- FIRST = false the current subarray is not the first subarray of the data array.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current subarray is the last subarray of the data array.
- LAST = false the current subarray is not the last subarray of the data array.

XMEAN (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_MVS_MISS (e.g. when FIRST=true), XMEAN is used as workspace to accumulate quantities on previous calls to COMP_MVS_MISS. XMEAN should not be changed between calls to COMP_MVS_MISS.

On exit, when LAST=true, XMEAN contains the mean values.

The shape of XMEAN must verify:

- size(XMEAN,1) = size(X,1)
- size(XMEAN,2) = size(X,2).

XVAR (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_MVS_MISS (e.g. when FIRST=true), XVAR is used as workspace to accumulate quantities on previous calls to COMP_MVS_MISS. XVAR should not be changed between calls to COMP_MVS_MISS.

On exit, when LAST=true, XVAR contains the variances.

The shape of XVAR must verify:

- size(XVAR,1) = size(X,1)
- size(XVAR,2) = size(X,2).

XSTD (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_MVS_MISS (e.g. when FIRST=true), XSTD is used as workspace to accumulate quantities on previous calls to COMP_MVS_MISS. XSTD should not be changed between calls to COMP_MVS_MISS.

On exit, when LAST=true, XSTD contains the standard deviations.

The shape of XSTD must verify:

- size(XSTD,1) = size(X,1)
- size(XSTD,2) = size(X,2).

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

XNOBS (OUTPUT, OPTIONAL) integer(i4b), dimension(:,:) On exit, XNOBS contains the numbers of non-missing observations on all variables. XNOBS needs to be specified only on the last call to COMP_MVS_MISS (e.g. when LAST=true).

The shape of XNOBS must verify:

- size(XNOBS,1) = size(X,1)
- size(XNOBS,2) = size(X,2).

Further Details

The subroutine computes the basic statistics with only one pass through the data.

If fewer than one valid (non-missing) observation is present for some variables, the pertinent statistics are set to XMISS.

6.22.16 subroutine comp_mvs (x, first, last, xmean, xvar, xstd, xmiss, xnobs)

Purpose

COMP_MVS_MISS computes estimates of means, variances and standard-deviations from a fourdimensional data array possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:,:,:) On entry, input subarray containing size(X,4) observations on size(X,1) by size(X,2) by size(X,3) variables from the array of data for which basic univariate statistics are desired. If all the data are available at once, X can be the full data array.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subarray is the first subarray of the data array.
- FIRST = false the current subarray is not the first subarray of the data array.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current subarray is the last subarray of the data array.
- LAST = false the current subarray is not the last subarray of the data array.

XMEAN (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, after the first call to COMP_MVS_MISS (e.g. when FIRST=true), XMEAN is used as workspace to accumulate quantities on previous calls to COMP_MVS_MISS. XMEAN should not be changed between calls to COMP_MVS_MISS.

On exit, when LAST=true, XMEAN contains the mean values.

The shape of XMEAN must verify:

- size(XMEAN,1) = size(X,1)
- size(XMEAN,2) = size(X,2)
- size(XMEAN,3) = size(X,3).

XVAR (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, after the first call to COMP_MVS_MISS (e.g. when FIRST=true), XVAR is used as workspace to accumulate quantities on previous calls to COMP_MVS_MISS. XVAR should not be changed between calls to COMP_MVS_MISS.

On exit, when LAST=true, XVAR contains the variances.

The shape of XVAR must verify:

- size(XVAR,1) = size(X,1)
- size(XVAR,2) = size(X,2)
- size(XVAR,3) = size(X,3).

XSTD (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, after the first call to COMP_MVS_MISS (e.g. when FIRST=true), XSTD is used as workspace to accumulate quantities on previous calls to COMP_MVS_MISS. XSTD should not be changed between calls to COMP_MVS_MISS.

On exit, when LAST=true, XSTD contains the standard deviations.

The shape of XSTD must verify:

- size(XSTD,1) = size(X,1)
- size(XSTD,2) = size(X,2)
- size(XSTD,3) = size(X,3).

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

XNOBS (OUTPUT, OPTIONAL) integer(i4b), dimension(:,:,:) On exit, XNOBS contains the numbers of non-missing observations on all variables. XNOBS needs to be specified only on the last call to COMP_MVS_MISS (e.g. when LAST=true).

The shape of XNOBS must verify:

- size(XNOBS,1) = size(X,1)
- size(XNOBS,2) = size(X,2)
- size(XNOBS,3) = size(X,3).

Further Details

The subroutine computes the basic statistics with only one pass through the data.

If fewer than one valid (non-missing) observation is present for some variables, the pertinent statistics are set to XMISS.

6.22.17 subroutine comp_mvs_grp (x, first, last, ngrp, ind, xmean_grp, xstd_grp, xn_grp)

Purpose

COMP_MVS_GRP computes estimates of univariate statistics by groups from a data vector.

Arguments

X (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X) observations from the vector of data for which univariate statistics by groups are desired. If all the data are available at once, X can be the full data vector.

FIRST (INPUT) logical(lgl) On entry, if:

- **FIRST = true the current subvector is the first subvector** of the data vector.
- **FIRST = false the current subvector is not the first subvector** of the data vector.

LAST (INPUT) logical(lgl) On entry, if:

- **LAST = true the current subvector is the last subvector** of the data vector.

- **LAST = false the current subvector is not the last subvector** of the data vector.

NGRP (INPUT) integer(i4b) On entry, the number of groups.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group in the analysis.

The size of IND must verify: size(IND) = size(X) .

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_MVS_GRP (e.g. when FIRST=true), XMEAN_GRP contains the mean values for the NGRP groups from previous calls to COMP_MVS_GRP. XMEAN_GRP should not be changed between calls to COMP_MVS_GRP.

On exit, when LAST=true, XMEAN_GRP contains the mean values for the NGRP groups of observations in the data vector.

The size of XMEAN_GRP must verify: size(XMEAN_GRP) = NGRP .

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_MVS_GRP (e.g. when FIRST=true), XSTD_GRP contains adjusted sums of squares for the NGRP groups from previous calls to COMP_MVS_GRP. XSTD_GRP should not be changed between calls to COMP_MVS_GRP.

On exit, when LAST=true, XSTD_GRP contains the standard deviations for the NGRP groups of observations in the data vector.

The size of XSTD_GRP must verify: size(XSTD_GRP) = NGRP .

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_MVS_GRP (e.g. when FIRST=true), XN_GRP contains counts of observations for the NGRP groups from previous calls to COMP_MVS_GRP. XN_GRP should not be changed between calls to COMP_MVS_GRP.

On exit, when LAST=true, XN_GRP contains the numbers of observations in the NGRP groups of observations in the data vector.

The size of XN_GRP must verify: size(XN_GRP) = NGRP .

Further Details

The subroutine computes all the statistics with only one pass through the data.

If fewer than one observation is present for some groups, the pertinent statistics are set to Nan code.

6.22.18 subroutine comp_mvs_grp (x, first, last, ngrp, ind, xmean_grp, xstd_grp, xn_grp, dimvar)

Purpose

COMP_MVS_GRP computes estimates of univariate statistics by groups from a data matrix.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which univariate statistics by

groups are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix.

FIRST (INPUT) logical(lgl) On entry, if:

- **FIRST = true the current submatrix is the first submatrix** of the data matrix.
- **FIRST = false the current submatrix is not the first submatrix** of the data matrix.

LAST (INPUT) logical(lgl) On entry, if:

- **LAST = true the current submatrix is the last submatrix** of the data matrix.
- **LAST = false the current submatrix is not the last submatrix** of the data matrix.

NGRP (INPUT) integer(i4b) On entry, the number of groups.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X,3-DIMVAR) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group in the analysis.

The size of IND must verify: size(IND) = size(X,3-DIMVAR) .

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_MVS_GRP (e.g. when FIRST=true), XMEAN_GRP contains the mean values for the NGRP groups from previous calls to COMP_MVS_GRP. XMEAN_GRP should not be changed between calls to COMP_MVS_GRP.

On exit, when LAST=true, XMEAN_GRP contains the mean values for the NGRP groups of observations on all the variables in the data matrix.

The shape of XMEAN_GRP must verify:

- size(XMEAN_GRP,1) = size(X,DIMVAR)
- size(XMEAN_GRP,2) = NGRP.

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_MVS_GRP (e.g. when FIRST=true), XSTD_GRP contains adjusted sums of squares for the NGRP groups from previous calls to COMP_MVS_GRP. XSTD_GRP should not be changed between calls to COMP_MVS_GRP.

On exit, when LAST=true, XSTD_GRP contains the standard deviations for the NGRP groups of observations on all the variables in the data matrix.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(X,DIMVAR)
- size(XSTD_GRP,2) = NGRP.

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_MVS_GRP (e.g. when FIRST=true), XN_GRP contains counts of observations for the NGRP groups from previous calls to COMP_MVS_GRP. XN_GRP should not be changed between calls to COMP_MVS_GRP.

On exit, when LAST=true, XN_GRP contains the numbers of observations in the NGRP groups for all the variables in the data matrix.

The size of XN_GRP must verify: size(XN_GRP) = NGRP .

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.

- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

Further Details

The subroutine computes all the statistics with only one pass through the data.

If fewer than one observation is present for some groups, the pertinent statistics are set to Nan code.

6.22.19 subroutine comp_mvs_grp (x, first, last, ngrp, ind, xmean_grp, xstd_grp, xn_grp)

Purpose

COMP_MVS_GRP computes estimates of univariate statistics by groups from a data tridimensional array.

Arguments

X (INPUT) real(stnd), dimension(:,:,:) On entry, input subarray containing size(X,3) observations on size(X,1) by size(X,2) variables from the array of data for which univariate statistics by groups are desired. If all the data are available at once, X can be the full data array.

FIRST (INPUT) logical(lgl) On entry, if:

- **FIRST = true the current subarray is the first subarray** of the data array.
- **FIRST = false the current subarray is not the first subarray** of the data array.

LAST (INPUT) logical(lgl) On entry, if:

- **LAST = true the current subarray is the last subarray** of the data array.
- **LAST = false the current subarray is not the last subarray** of the data array.

NGRP (INPUT) integer(i4b) On entry, the number of groups.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X,3) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group in the analysis.

The size of IND must verify: size(IND) = size(X,3) .

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, after the first call to COMP_MVS_GRP (e.g. when FIRST=true), XMEAN_GRP contains the mean values for the NGRP groups from previous calls to COMP_MVS_GRP. XMEAN_GRP should not be changed between calls to COMP_MVS_GRP.

On exit, when LAST=true, XMEAN_GRP contains the mean values for the NGRP groups of observations on all the variables in the data array.

The shape of XMEAN_GRP must verify:

- size(XMEAN_GRP,1) = size(X,1)
- size(XMEAN_GRP,2) = size(X,2)
- size(XMEAN_GRP,3) = NGRP.

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, after the first call to COMP_MVS_GRP (e.g. when FIRST=true), XSTD_GRP contains adjusted sums of squares for the NGRP groups from previous calls to COMP_MVS_GRP. XSTD_GRP should not be changed between calls to COMP_MVS_GRP.

On exit, when LAST=true, XSTD_GRP contains the standard deviations for the NGRP groups of observations on all the variables in the data array.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(X,1)
- size(XSTD_GRP,2) = size(X,2)
- size(XSTD_GRP,3) = NGRP.

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_MVS_GRP (e.g. when FIRST=true), XN_GRP contains counts of observations for the NGRP groups from previous calls to COMP_MVS_GRP. XN_GRP should not be changed between calls to COMP_MVS_GRP.

On exit, when LAST=true, XN_GRP contains the numbers of observations in the NGRP groups for all the variables in the data array.

The size of XN_GRP must verify: size(XN_GRP) = NGRP .

Further Details

The subroutine computes all the statistics with only one pass through the data.

If fewer than one observation is present for some groups, the pertinent statistics are set to Nan code.

6.22.20 subroutine comp_mvs_grp (x, first, last, ngrp, ind, xmean_grp, xstd_grp, xn_grp)

Purpose

COMP_MVS_GRP computes estimates of univariate statistics by groups from a data fourdimensional array.

Arguments

X (INPUT) real(stnd), dimension(:,:,:,:) On entry, input subarray containing size(X,4) observations on size(X,1) by size(X,2) by size(X,3) variables from the array of data for which univariate statistics by groups are desired. If all the data are available at once, X can be the full data array.

FIRST (INPUT) logical(lgl) On entry, if:

- **FIRST = true the current subarray is the first subarray** of the data array.
- **FIRST = false the current subarray is not the first subarray** of the data array.

LAST (INPUT) logical(lgl) On entry, if:

- **LAST = true the current subarray is the last subarray** of the data array.
- **LAST = false the current subarray is not the last subarray** of the data array.

NGRP (INPUT) integer(i4b) On entry, the number of groups.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X,4) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group in the analysis.

The size of IND must verify: size(IND) = size(X,4) .

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:,:) On entry, after the first call to COMP_MVS_GRP (e.g. when FIRST=true), XMEAN_GRP contains the mean values for the NGRP groups from previous calls to COMP_MVS_GRP. XMEAN_GRP should not be changed between calls to COMP_MVS_GRP.

On exit, when LAST=true, XMEAN_GRP contains the mean values for the NGRP groups of observations on all the variables in the data array.

The shape of XMEAN_GRP must verify:

- size(XMEAN_GRP,1) = size(X,1)
- size(XMEAN_GRP,2) = size(X,2)
- size(XMEAN_GRP,3) = size(X,3)
- size(XMEAN_GRP,4) = NGRP.

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:,:) On entry, after the first call to COMP_MVS_GRP (e.g. when FIRST=true), XSTD_GRP contains adjusted sums of squares for the NGRP groups from previous calls to COMP_MVS_GRP. XSTD_GRP should not be changed between calls to COMP_MVS_GRP.

On exit, when LAST=true, XSTD_GRP contains the standard deviations for the NGRP groups of observations on all the variables in the data array.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(X,1),
- size(XSTD_GRP,2) = size(X,2)
- size(XSTD_GRP,3) = size(X,3)
- size(XSTD_GRP,4) = NGRP.

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_MVS_GRP (e.g. when FIRST=true), XN_GRP contains counts of observations for the NGRP groups from previous calls to COMP_MVS_GRP. XN_GRP should not be changed between calls to COMP_MVS_GRP.

On exit, when LAST=true, XN_GRP contains the numbers of observations in the NGRP groups for all the variables in the data array.

The size of XN_GRP must verify: size(XN_GRP) = NGRP .

Further Details

The subroutine computes all the statistics with only one pass through the data.

If fewer than one observation is present for some groups, the pertinent statistics are set to Nan code.

6.22.21 subroutine comp_mvs_grp (x, first, last, ngrp, ind, xmean_grp, xstd_grp, xn_grp, xmiss)

Purpose

COMP_MVS_GRP_MISS computes estimates of univariate statistics by groups from a data vector possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X) observations from the vector of data for which univariate statistics by groups are desired. If all the data are available at once, X can be the full data vector.

FIRST (INPUT) logical(lgl) On entry, if:

- **FIRST = true the current subvector is the first subvector** of the data vector.
- **FIRST = false the current subvector is not the first subvector** of the data vector.

LAST (INPUT) logical(lgl) On entry, if:

- **LAST = true the current subvector is the last subvector** of the data vector.
- **LAST = false the current subvector is not the last subvector** of the data vector.

NGRP (INPUT) integer(i4b) On entry, the number of groups.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group in the analysis.

The size of IND must verify: size(IND) = size(X) .

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_MVS_GRP_MISS (e.g. when FIRST=true), XMEAN_GRP contains the mean values for the NGRP groups from previous calls to COMP_MVS_GRP_MISS. XMEAN_GRP should not be changed between calls to COMP_MVS_GRP_MISS.

On exit, when LAST=true, XMEAN_GRP contains the mean values for the NGRP groups of observations in the data vector.

The size of XMEAN_GRP must verify: size(XMEAN_GRP) = NGRP .

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_MVS_GRP_MISS (e.g. when FIRST=true), XSTD_GRP contains adjusted sums of squares for the NGRP groups from previous calls to COMP_MVS_GRP_MISS. XSTD_GRP should not be changed between calls to COMP_MVS_GRP_MISS.

On exit, when LAST=true, XSTD_GRP contains the standard deviations for the NGRP groups of observations in the data vector.

The size of XSTD_GRP must verify: size(XSTD_GRP) = NGRP .

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_MVS_GRP_MISS (e.g. when FIRST=true), XN_GRP contains counts of non-missing observations for the NGRP groups from previous calls to COMP_MVS_GRP_MISS. XN_GRP should not be changed between calls to COMP_MVS_GRP_MISS.

On exit, XN_GRP contains the number of non-missing observations in the NGRP groups of observations in the data vector.

The size of XN_GRP must verify: size(XN_GRP) = NGRP .

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

Further Details

The subroutine computes all the statistics with only one pass through the data.

If fewer than one valid observation were present for some groups of observations, the pertinent statistics are set to missing (XMISS value).

6.22.22 subroutine comp_mvs_grp (x, first, last, ngrp, ind, xmean_grp, xstd_grp, xn_grp, xmiss, dimvar)

Purpose

COMP_MVS_GRP_MISS computes estimates of univariate statistics by groups from a data matrix possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which univariate statistics by groups are desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix.

FIRST (INPUT) logical(lgl) On entry, if:

- **FIRST = true the current submatrix is the first submatrix** of the data matrix.
- **FIRST = false the current submatrix is not the first submatrix** of the data matrix.

LAST (INPUT) logical(lgl) On entry, if:

- **LAST = true the current submatrix is the last submatrix** of the data matrix.
- **LAST = false the current submatrix is not the last submatrix** of the data matrix.

NGRP (INPUT) integer(i4b) On entry, the number of groups.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X,3-DIMVAR) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group in the analysis.

The size of IND must verify: size(IND) = size(X,3-DIMVAR) .

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_MVS_GRP_MISS (e.g. when FIRST=true), XMEAN_GRP contains the mean values for the NGRP groups from previous calls to COMP_MVS_GRP_MISS. XMEAN_GRP should not be changed between calls to COMP_MVS_GRP_MISS.

On exit, when LAST=true, XMEAN_GRP contains the mean values for the NGRP groups of observations on all the variables in the data matrix.

The shape of XMEAN_GRP must verify:

- size(XMEAN_GRP,1) = size(X,DIMVAR)
- size(XMEAN_GRP,2) = NGRP.

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_MVS_GRP_MISS (e.g. when FIRST=true), XSTD_GRP contains adjusted sums of squares for the NGRP groups from previous calls to COMP_MVS_GRP_MISS. XSTD_GRP should not be changed between calls to COMP_MVS_GRP_MISS.

On exit, when LAST=true, XSTD_GRP contains the standard deviations for the NGRP groups of observations on all the variables in the data matrix.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(X,DIMVAR)
- size(XSTD_GRP,2) = NGRP.

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_MVS_GRP_MISS (e.g. when FIRST=true), XN_GRP contains counts of non-missing observations for the NGRP groups from previous calls to COMP_MVS_GRP_MISS. XN_GRP should not be changed between calls to COMP_MVS_GRP_MISS.

On exit, XN_GRP contains the numbers of non-missing observations in the NGRP groups for all the variables in the data matrix.

The shape of XN_GRP must verify:

- size(XN_GRP,1) = size(X,DIMVAR)
- size(XN_GRP,2) = NGRP.

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

Further Details

The subroutine computes all the statistics with only one pass through the data.

If fewer than one valid observation were present for some variables and/or groups of observations, the pertinent statistics are set to missing (XMISS value).

6.22.23 subroutine comp_mvs_grp (x, first, last, ngrp, ind, xmean_grp, xstd_grp, xn_grp, xmiss)

Purpose

COMP_MVS_GRP_MISS computes estimates of univariate statistics by groups from a data tridimensional array possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:,:) On entry, input subarray containing size(X,3) observations on size(X,1) by size(X,2) variables from the array of data for which univariate statistics by groups are desired. If all the data are available at once, X can be the full data array.

FIRST (INPUT) logical(lgl) On entry, if:

- **FIRST = true the current subarray is the first subarray** of the data array.

- **FIRST = false the current subarray is not the first subarray** of the data array.

LAST (INPUT) logical(lgl) On entry, if:

- **LAST = true the current subarray is the last subarray** of the data array.
- **LAST = false the current subarray is not the last subarray** of the data array.

NGRP (INPUT) integer(i4b) On entry, the number of groups.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X,3) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group in the analysis.

The size of IND must verify: size(IND) = size(X,3) .

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, after the first call to COMP_MVS_GRP_MISS (e.g. when FIRST=true), XMEAN_GRP contains the mean values for the NGRP groups from previous calls to COMP_MVS_GRP_MISS. XMEAN_GRP should not be changed between calls to COMP_MVS_GRP_MISS.

On exit, when LAST=true, XMEAN_GRP contains the mean values for the NGRP groups of observations on all the variables in the data array.

The shape of XMEAN_GRP must verify:

- size(XMEAN_GRP,1) = size(X,1)
- size(XMEAN_GRP,2) = size(X,2)
- size(XMEAN_GRP,3) = NGRP.

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, after the first call to COMP_MVS_GRP_MISS (e.g. when FIRST=true), XSTD_GRP contains adjusted sums of squares for the NGRP groups from previous calls to COMP_MVS_GRP_MISS. XSTD_GRP should not be changed between calls to COMP_MVS_GRP_MISS.

On exit, when LAST=true, XSTD_GRP contains the standard deviations for the NGRP groups of observations on all the variables in the data array.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(X,1)
- size(XSTD_GRP,2) = size(X,2)
- size(XSTD_GRP,3) = NGRP.

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, after the first call to COMP_MVS_GRP_MISS (e.g. when FIRST=true), XN_GRP contains counts of non-missing observations for the NGRP groups from previous calls to COMP_MVS_GRP_MISS. XN_GRP should not be changed between calls to COMP_MVS_GRP_MISS.

On exit, when LAST=true, XN_GRP contains the numbers of non-missing observations in the NGRP groups for all the variables in the data array.

The shape of XN_GRP must verify:

- size(XN_GRP,1) = size(X,1)
- size(XN_GRP,2) = size(X,2)
- size(XN_GRP,3) = NGRP.

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

Further Details

The subroutine computes all the statistics with only one pass through the data.

If fewer than one valid observation were present for some variables and/or groups of observations, the pertinent statistics are set to missing (XMISS value).

6.22.24 subroutine comp_mvs_grp (x, first, last, ngrp, ind, xmean_grp, xstd_grp, xn_grp, xmiss)

Purpose

COMP_MVS_GRP_MISS computes estimates of univariate statistics by groups from a data fourdimensional array possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:,:,:) On entry, input subarray containing size(X,4) observations on size(X,1) by size(X,2) by size(X,3) variables from the array of data for which univariate statistics by groups are desired. If all the data are available at once, X can be the full data array.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subarray is the first subarray of the data array.
- FIRST = false the current subarray is not the first subarray of the data array.

LAST (INPUT) logical(lgl) On entry, if:

- **LAST = true the current subarray is the last subarray** of the data array.
- **LAST = false the current subarray is not the last subarray** of the data array.

NGRP (INPUT) integer(i4b) On entry, the number of groups.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X,4) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group in the analysis.

The size of IND must verify: size(IND) = size(X,4) .

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:,:) On entry, after the first call to COMP_MVS_GRP_MISS (e.g. when FIRST=true), XMEAN_GRP contains the mean values for the NGRP groups from previous calls to COMP_MVS_GRP_MISS. XMEAN_GRP should not be changed between calls to COMP_MVS_GRP_MISS.

On exit, when LAST=true, XMEAN_GRP contains the mean values for the NGRP groups of observations on all the variables in the data array.

The shape of XMEAN_GRP must verify:

- size(XMEAN_GRP,1) = size(X,1)
- size(XMEAN_GRP,2) = size(X,2)
- size(XMEAN_GRP,3) = size(X,3)
- size(XMEAN_GRP,4) = NGRP.

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:,:) On entry, after the first call to COMP_MVS_GRP_MISS (e.g. when FIRST=true), XSTD_GRP contains adjusted sums of squares for the NGRP groups from previous calls to COMP_MVS_GRP_MISS. XSTD_GRP should not be changed between calls to COMP_MVS_GRP_MISS.

On exit, when LAST=true, XSTD_GRP contains the standard deviations for the NGRP groups of observations on all the variables in the data array.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(X,1)
- size(XSTD_GRP,2) = size(X,2)
- size(XSTD_GRP,3) = size(X,3)
- size(XSTD_GRP,4) = NGRP.

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:,:) On entry, after the first call to COMP_MVS_GRP_MISS (e.g. when FIRST=true), XN_GRP contains counts of non-missing observations for the NGRP groups from previous calls to COMP_MVS_GRP_MISS. XN_GRP should not be changed between calls to COMP_MVS_GRP_MISS.

On exit, when LAST=true, XN_GRP contains the numbers of non-missing observations in the NGRP groups for all the variables in the data array.

The shape of XN_GRP must verify:

- size(XN_GRP,1) = size(X,1)
- size(XN_GRP,2) = size(X,2)
- size(XN_GRP,3) = size(X,3)
- size(XN_GRP,4) = NGRP.

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

Further Details

The subroutine computes all the statistics with only one pass through the data.

If fewer than one valid observation were present for some variables and/or groups of observations, the pertinent statistics are set to missing (XMISS value).

6.22.25 `subroutine update_mvs ( xmean, xvar, xnobs, xmean2, xvar2, xnobs2 )`

Purpose

UPDATE_MVS computes sample mean and corrected sum of squares for a sample of size XNOBS+XNOBS2 given the means and corrected sums of squares for two subsamples of size XNOBS and XNOBS2 as output by a call to COMP_MVS when LAST=false on the two subsamples separetely.

The sample means, standard-deviations for the sample of size XNOBS+XNOBS2 may be obtained by a call to COMP_MVS with LAST=true.

Arguments

XMEAN (INPUT/OUTPUT) real(stnd) On entry, the sample mean of the first sample of size XNOBS.

On exit, the sample mean of the combined sample of size XNOBS+XNOBS2.

XVAR (INPUT/OUTPUT) real(stnd) On entry, the corrected sum of squares of the first sample of size XNOBS.

On exit, the corrected sum of squares of the combined sample of size XNOBS+XNOBS2.

XNOBS (INPUT/OUTPUT) real(stnd) On entry, the number of observations of the first sample.

On exit, the number of observations of the combined sample (i.e. XNOBS+XNOBS2).

XMEAN2 (INPUT) real(stnd) On entry, the sample mean of the second sample of size XNOBS2.

XVAR2 (INPUT) real(stnd) On entry, the corrected sum of squares of the second sample of size XNOBS2.

XNOBS2 (INPUT) real(stnd) On entry, the number of observations of the second sample.

Further Details

One possible application of this subroutine is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares computed for each subsample independently using COMP_MVS. The means and corrected sums of squares for the original sample can then be calculated using UPDATE_MVS. The means, variances and standard-deviations for the original sample can be computed by a final call to COMP_MVS with LAST=true.

This subroutine is adapted from:

(1) **Chan, T.F., Golub, G.H., and Leveque, R.J., 1979:** Updating formulae and a pairwise algorithm for computing sample variances. STAN-CS-79-773, November 1979.

6.22.26 `subroutine update_mvs ( xmean, xvar, xnobs, xmean2, xvar2, xnobs2 )`

Purpose

UPDATE_MVS computes sample means and corrected sums of squares by groups for a sample of size XNOBS+XNOBS2 given the means and corrected sums of squares for two subsamples of size XNOBS and XNOBS2 as output by a call to COMP_MVS when LAST=false on the two subsamples separetely.

The sample means, standard-deviations for the sample of size XNOBS+XNOBS2 may be obtained by a call to COMP_MVS with LAST=true.

Arguments

XMEAN (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the sample means of the first sample of size XNOBS.

On exit, the sample means of the combined sample of size XNOBS+XNOBS2.

XVAR (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the corrected sums of squares of the first sample of size XNOBS.

On exit, the corrected sums of squares of the combined sample of size XNOBS+XNOBS2.

The shape of XVAR must verify: size(XVAR) = size(XMEAN).

XNOBS (INPUT/OUTPUT) real(stnd) On entry, the number of observations of the first sample.

On exit, the number of observations of the combined sample (i.e. XNOBS+XNOBS2).

XMEAN2 (INPUT) real(stnd), dimension(:) On entry, the sample means of the second sample of size XNOBS2.

The shape of XMEAN2 must verify: size(XMEAN2) = size(XMEAN).

XVAR2 (INPUT) real(stnd), dimension(:) On entry, the corrected sum of squares of the second sample of size XNOBS2.

The shape of XVAR2 must verify: size(XVAR2) = size(XMEAN).

XNOBS2 (INPUT) real(stnd) On entry, the number of observations of the second sample.

Further Details

One possible application of this subroutine is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares computed for each subsample independently using COMP_MVS. The means and corrected sums of squares for the original sample can then be calculated using UPDATE_MVS. The means, variances and standard-deviations for the original sample can be computed by a final call to COMP_MVS with LAST=true.

This subroutine is adapted from:

(1) **Chan, T.F., Golub, G.H., and Leveque, R.J., 1979:** Updating formulae and a pairwise algorithm for computing sample variances. STAN-CS-79-773, November 1979.

6.22.27 `subroutine update_mvs ( xmean, xvar, xnobs, xmean2, xvar2, xnobs2 )`

Purpose

UPDATE_MVS computes sample means and corrected sums of squares by groups for a sample of size XNOBS+XNOBS2 given the means and corrected sums of squares for two subsamples of size XNOBS and XNOBS2 as output by a call to COMP_MVS when LAST=false on the two subsamples separetely.

The sample means, standard-deviations for the sample of size XNOBS+XNOBS2 may be obtained by a call to COMP_MVS with LAST=true.

Arguments

XMEAN (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the sample means of the first sample of size XNOBS.

On exit, the sample means of the combined sample of size XNOBS+XNOBS2.

XVAR (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the corrected sums of squares of the first sample of size XNOBS.

On exit, the corrected sums of squares of the combined sample of size XNOBS+XNOBS2

The shape of XVAR must verify:

- size(XVAR,1) = size(XMEAN,1)
- size(XVAR,2) = size(XMEAN,2).

XNOBS (INPUT/OUTPUT) real(stnd) On entry, the number of observations of the first sample.

On exit, the number of observations of the combined sample (i.e. XNOBS+XNOBS2).

XMEAN2 (INPUT) real(stnd), dimension(:,:) On entry, the sample means of the second sample of size XNOBS2.

The shape of XMEAN2 must verify:

- size(XMEAN2,1) = size(XMEAN,1)
- size(XMEAN2,2) = size(XMEAN,2).

XVAR2 (INPUT) real(stnd), dimension(:,:) On entry, the corrected sum of squares of the second sample of size XNOBS2.

The shape of XVAR2 must verify:

- size(XVAR2,1) = size(XMEAN,1)
- size(XVAR2,2) = size(XMEAN,2).

XNOBS2 (INPUT) real(stnd), dimension(:) On entry, the number of observations of the second sample.

Further Details

One possible application of this subroutine is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares computed for each subsample independently using COMP_MVS. The means and corrected sums of squares for the original sample can then be calculated using UPDATE_MVS. The means, variances and standard-deviations for the original sample can be computed by a final call to COMP_MVS with LAST=true.

This subroutine is adapted from:

(1) **Chan, T.F., Golub, G.H., and Leveque, R.J., 1979:** Updating formulae and a pairwise algorithm for computing sample variances. STAN-CS-79-773, November 1979.

6.22.28 `subroutine update_mvs ( xmean, xvar, xnobs, xmean2, xvar2, xnobs2 )`

Purpose

UPDATE_MVS computes sample means and corrected sums of squares by groups for a sample of size XNOBS+XNOBS2 given the means and corrected sums of squares for two subsamples of size XNOBS and XNOBS2 as output by a call to COMP_MVS when LAST=false on the two subsamples separetely.

The sample means, standard-deviations for the sample of size XNOBS+XNOBS2 may be obtained by a call to COMP_MVS with LAST=true.

Arguments

XMEAN (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, the sample means of the first sample of size XNOBS.

On exit, the sample means of the combined sample of size XNOBS+XNOBS2.

XVAR (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, the corrected sums of squares of the first sample of size XNOBS.

On exit, the corrected sums of squares of the combined sample of size XNOBS+XNOBS2.

The shape of XVAR must verify:

- size(XVAR,1) = size(XMEAN,1)
- size(XVAR,2) = size(XMEAN,2)
- size(XVAR,3) = size(XMEAN,3).

XNOBS (INPUT/OUTPUT) real(stnd) On entry, the number of observations of the first sample.

On exit, the number of observations of the combined sample (i.e. XNOBS+XNOBS2).

XMEAN2 (INPUT) real(stnd), dimension(:,:,:) On entry, the sample means of the second sample of size XNOBS2.

The shape of XMEAN2 must verify:

- size(XMEAN2,1) = size(XMEAN,1)
- size(XMEAN2,2) = size(XMEAN,2)
- size(XMEAN2,3) = size(XMEAN,3).

XVAR2 (INPUT) real(stnd), dimension(:,:,:) On entry, the corrected sum of squares of the second sample of size XNOBS2.

The shape of XVAR2 must verify:

- size(XVAR2,1) = size(XMEAN,1)
- size(XVAR2,2) = size(XMEAN,2)
- size(XVAR2,3) = size(XMEAN,3).

XNOBS2 (INPUT) real(stnd) On entry, the number of observations of the second sample.

Further Details

One possible application of this subroutine is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares computed for each subsample independently using COMP_MVS. The means and corrected sums of squares for the original sample can then be calculated using UPDATE_MVS. The means, variances and standard-deviations for the original sample can be computed by a final call to COMP_MVS with LAST=true.

This subroutine is adapted from:

(1) **Chan, T.F., Golub, G.H., and Leveque, R.J., 1979:** Updating formulae and a pairwise algorithm for computing sample variances. STAN-CS-79-773, November 1979.

6.22.29 subroutine update_mvs_grp (xmean_grp, xstd_grp, xn_grp, xmean_grp2, xstd_grp2, xn_grp2)

Purpose

UPDATE_MVS_GRP computes sample means and corrected sums of squares by groups for a sample of size sum(XN_GRP)+sum(XN_GRP2) given the means and corrected sums of squares for two subsamples of size sum(XN_GRP) and sum(XN_GRP2) as output by a call to COMP_MVS_GRP when LAST=false on the two subsamples separetely.

The sample means, variances and standard-deviations for the sample of size sum(XN_GRP)+sum(XN_GRP2) may be obtained by a call to COMP_MVS_GRP with LAST=true.

Arguments

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the sample means for the groups computed on the first sample.

On exit, the sample means for the groups computed on the combined sample.

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the sample corrected sums of squares for the groups computed on the first sample.

On exit, the sample corrected sums of squares for the groups computed on the combined sample.

The shape of XSTD_GRP must verify: size(XSTD_GRP) = size(XMEAN_GRP).

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the number of observations in each group in the first sample.

On exit, the number of observations in each group in the combined sample.

The size of XN_GRP must verify: size(XN_GRP) = size(XMEAN_GRP) .

XMEAN_GRP2 (INPUT) real(stnd), dimension(:) On entry, the sample means for the groups computed on the second sample.

The shape of XMEAN_GRP2 must verify: size(XMEAN_GRP2) = size(XMEAN_GRP).

XSTD_GRP2 (INPUT) real(stnd), dimension(:) On entry, the sample corrected sums of squares for the groups computed on the second sample.

The shape of XSTD_GRP2 must verify: size(XSTD_GRP2) = size(XMEAN_GRP).

XN_GRP2 (INPUT) real(stnd), dimension(:) On entry, the number of observations in each group in the second sample.

The size of XN_GRP2 must verify: size(XN_GRP2) = size(XMEAN_GRP) .

Further Details

One possible application of this subroutine is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares computed for each subsample independently using COMP_MVS_GRP. The means and corrected sums of squares for the original sample can then be calculated using UPDATE_MVS_GRP.

The means, variances and standard-deviations for the original sample can be computed by a final call to COMP_MVS_GRP with LAST=true.

This subroutine is adapted from:

(1) **Chan, T.F., Golub, G.H., and Leveque, R.J., 1979:** Updating formulae and a pairwise algorithm for computing sample variances. STAN-CS-79-773, November 1979.

6.22.30 subroutine update_mvs_grp (xmean_grp, xstd_grp, xn_grp, xmean_grp2, xstd_grp2, xn_grp2)

Purpose

UPDATE_MVS_GRP computes sample means and corrected sums of squares by groups for a sample of size sum(XN_GRP)+sum(XN_GRP2) given the means and corrected sums of squares for two subsamples of size sum(XN_GRP) and sum(XN_GRP2) as output by a call to COMP_MVS_GRP when LAST=false on the two subsamples separetely.

The sample means, variances and standard-deviations for the sample of size sum(XN_GRP)+sum(XN_GRP2) may be obtained by a call to COMP_MVS_GRP with LAST=true.

Arguments

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the sample means for the groups computed on the first sample vector.

On exit, the sample means for the groups computed on the combined sample vector.

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the sample corrected sums of squares for the groups computed on the first sample vector.

On exit, the sample corrected sums of squares for the groups computed on the combined sample vector.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(XMEAN_GRP,1)
- size(XSTD_GRP,2) = size(XMEAN_GRP,2).

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the number of observations in each group in the first sample vector.

On exit, the number of observations in each group in the combined sample vector.

The size of XN_GRP must verify: size(XN_GRP) = size(XMEAN_GRP,2) .

XMEAN_GRP2 (INPUT) real(stnd), dimension(:,:) On entry, the sample means for the groups computed on the second sample vector.

The shape of XMEAN_GRP2 must verify:

- size(XMEAN_GRP2,1) = size(XMEAN_GRP,1)
- size(XMEAN_GRP2,2) = size(XMEAN_GRP,2)

XSTD_GRP2 (INPUT) real(stnd), dimension(:,:) On entry, the sample corrected sums of squares for the groups computed on the second sample vector.

The shape of XSTD_GRP2 must verify:

- size(XSTD_GRP2,1) = size(XMEAN_GRP,1)
- size(XSTD_GRP2,2) = size(XMEAN_GRP,2).

XN_GRP2 (INPUT) real(stnd), dimension(:) On entry, the number of observations in each group in the second sample vector.

The size of XN_GRP2 must verify: size(XN_GRP2) = size(XMEAN_GRP,2) .

Further Details

One possible application of this subroutine is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares computed for each subsample independently using COMP_MVS_GRP. The means and corrected sums of squares for the original sample can then be calculated using UPDATE_MVS_GRP.

The means, variances and standard-deviations for the original sample can be computed by a final call to COMP_MVS_GRP with LAST=true.

This subroutine is adapted from:

(1) **Chan, T.F., Golub, G.H., and Leveque, R.J., 1979:** Updating formulae and a pairwise algorithm for computing sample variances. STAN-CS-79-773, November 1979.

6.22.31 `subroutine update_mvs_grp ( xmean_grp, xstd_grp, xn_grp, xmean_grp2, xstd_grp2, xn_grp2 )`

Purpose

UPDATE_MVS_GRP computes sample means and corrected sums of squares by groups for a sample of size sum(XN_GRP)+sum(XN_GRP2) given the means and corrected sums of squares for two subsamples of size sum(XN_GRP) and sum(XN_GRP2) as output by a call to COMP_MVS_GRP when LAST=false on the two subsamples separetely.

The sample means, variances and standard-deviations for the sample of size sum(XN_GRP)+sum(XN_GRP2) may be obtained by a call to COMP_MVS_GRP with LAST=true.

Arguments

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, the sample means for the groups computed on the first sample matrix.

On exit, the sample means for the groups computed on the combined sample matrix.

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, the sample corrected sums of squares for the groups computed on the first sample matrix.

On exit, the sample corrected sums of squares for the groups computed on the combined sample matrix.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(XMEAN_GRP,1)
- size(XSTD_GRP,2) = size(XMEAN_GRP,2)
- size(XSTD_GRP,3) = size(XMEAN_GRP,3).

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the number of observations in each group in the first sample matrix.

On exit, the number of observations in each group in the combined sample matrix.

The size of XN_GRP must verify: size(XN_GRP) = size(XMEAN_GRP,3) .

XMEAN_GRP2 (INPUT) real(stnd), dimension(:,:,:) On entry, the sample means for the groups computed on the second sample matrix.

The shape of XMEAN_GRP2 must verify:

- size(XMEAN_GRP2,1) = size(XMEAN_GRP,1)
- size(XMEAN_GRP2,2) = size(XMEAN_GRP,2)
- size(XMEAN_GRP2,3) = size(XMEAN_GRP,3).

XSTD_GRP2 (INPUT) real(stnd), dimension(:,:,:) On entry, the sample corrected sums of squares for the groups computed on the second sample matrix.

The shape of XSTD_GRP2 must verify:

- size(XSTD_GRP2,1) = size(XMEAN_GRP,1)
- size(XSTD_GRP2,2) = size(XMEAN_GRP,2)
- size(XSTD_GRP2,3) = size(XMEAN_GRP,3).

XN_GRP2 (INPUT) real(stnd), dimension(:) On entry, the number of observations in each group in the second sample matrix.

The size of XN_GRP2 must verify: size(XN_GRP2) = size(XMEAN_GRP,3) .

Further Details

One possible application of this subroutine is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares computed for each subsample independently using COMP_MVS_GRP. The means and corrected sums of squares for the original sample can then be calculated using UPDATE_MVS_GRP.

The means, variances and standard-deviations for the original sample can be computed by a final call to COMP_MVS_GRP with LAST=true.

This subroutine is adapted from:

(1) **Chan, T.F., Golub, G.H., and Leveque, R.J., 1979:** Updating formulae and a pairwise algorithm for computing sample variances. STAN-CS-79-773, November 1979.

6.22.32 `subroutine update_mvs_grp ( xmean_grp, xstd_grp, xn_grp, xmean_grp2, xstd_grp2, xn_grp2 )`

Purpose

UPDATE_MVS_GRP computes sample means and corrected sums of squares by groups for a sample of size sum(XN_GRP)+sum(XN_GRP2) given the means and corrected sums of squares for two subsamples of size sum(XN_GRP) and sum(XN_GRP2) as output by a call to COMP_MVS_GRP when LAST=false on the two subsamples separetely.

The sample means, variances and standard-deviations for the sample of size sum(XN_GRP)+sum(XN_GRP2) may be obtained by a call to COMP_MVS_GRP with LAST=true.

Arguments

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:,:) On entry, the sample means for the groups computed on the first sample array.

On exit, the sample means for the groups computed on the combined sample array.

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:,:) On entry, the sample corrected sums of squares for the groups computed on the first sample array.

On exit, the sample corrected sums of squares for the groups computed on the combined sample array.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(XMEAN_GRP,1)
- size(XSTD_GRP,2) = size(XMEAN_GRP,2)
- size(XSTD_GRP,3) = size(XMEAN_GRP,3)
- size(XSTD_GRP,4) = size(XMEAN_GRP,4).

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the number of observations in each group in the first sample array.

On exit, the number of observations in each group in the combined sample array.

The size of XN_GRP must verify: size(XN_GRP) = size(XMEAN_GRP,4) .

XMEAN_GRP2 (INPUT) real(stnd), dimension(:,:,:,:) On entry, the sample means for the groups computed on the second sample array.

The shape of XMEAN_GRP2 must verify:

- size(XMEAN_GRP2,1) = size(XMEAN_GRP,1)
- size(XMEAN_GRP2,2) = size(XMEAN_GRP,2)
- size(XMEAN_GRP2,3) = size(XMEAN_GRP,3)
- size(XMEAN_GRP2,4) = size(XMEAN_GRP,4).

XSTD_GRP2 (INPUT) real(stnd), dimension(:,:,:,:) On entry, the sample corrected sums of squares for the groups computed on the second sample array.

The shape of XSTD_GRP2 must verify:

- size(XSTD_GRP2,1) = size(XMEAN_GRP,1)
- size(XSTD_GRP2,2) = size(XMEAN_GRP,2)
- size(XSTD_GRP2,3) = size(XMEAN_GRP,3)
- size(XSTD_GRP2,4) = size(XMEAN_GRP,4).

XN_GRP2 (INPUT) real(stnd), dimension(:) On entry, the number of observations in each group in the second sample array.

The size of XN_GRP2 must verify: size(XN_GRP2) = size(XMEAN_GRP,4) .

Further Details

One possible application of this subroutine is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares

computed for each subsample independently using COMP_MVS_GRP. The means and corrected sums of squares for the original sample can then be calculated using UPDATE_MVS_GRP.

The means, variances and standard-deviations for the original sample can be computed by a final call to COMP_MVS_GRP with LAST=true.

This subroutine is adapted from:

(1) **Chan, T.F., Golub, G.H., and Leveque, R.J., 1979:** Updating formulae and a pairwise algorithm for computing sample variances. STAN-CS-79-773, November 1979.

6.22.33 subroutine update_mvs_grp (xmean_grp, xstd_grp, xn_grp, xmean_grp2, xstd_grp2, xn_grp2)

Purpose

UPDATE_MVS_GRP_MISS computes sample means and corrected sums of squares by groups for a sample, possibly containing missing values, given the means and corrected sums of squares for two subsamples as output by a call to COMP_MVS_GRP_MISS when LAST=false on the two subsamples separetely.

The sample means, variances and standard-deviations for the sample may be obtained by a call to COMP_MVS_GRP_MISS with LAST=true.

Arguments

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the sample means for the groups computed on the first sample vector.

On exit, the sample means for the groups computed on the combined sample vector.

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the sample corrected sums of squares for the groups computed on the first sample vector.

On exit, the sample corrected sums of squares for the groups computed on the combined sample vector.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(XMEAN_GRP,1)
- size(XSTD_GRP,2) = size(XMEAN_GRP,2).

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the number of observations in each group in the first sample vector.

On exit, the number of observations in each group in the combined sample vector.

The shape of XSTD_GRP must verify:

- size(XN_GRP,1) = size(XMEAN_GRP,1)
- size(XN_GRP,2) = size(XMEAN_GRP,2).

XMEAN_GRP2 (INPUT) real(stnd), dimension(:,:) On entry, the sample means for the groups computed on the second sample vector.

The shape of XMEAN_GRP2 must verify:

- size(XMEAN_GRP2,1) = size(XMEAN_GRP,1)
- size(XMEAN_GRP2,2) = size(XMEAN_GRP,2).

XSTD_GRP2 (INPUT) real(stnd), dimension(:,:) On entry, the sample corrected sums of squares for the groups computed on the second sample vector.

The shape of XSTD_GRP2 must verify:

- size(XSTD_GRP2,1) = size(XMEAN_GRP,1)
- size(XSTD_GRP2,2) = size(XMEAN_GRP,2).

XN_GRP2 (INPUT) real(stnd), dimension(:,:) On entry, the number of observations in each group in the second sample vector.

The shape of XSTD_GRP must verify:

- size(XN_GRP2,1) = size(XMEAN_GRP,1)
- size(XN_GRP2,2) = size(XMEAN_GRP,2).

Further Details

One possible application of this subroutine is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares computed for each subsample independently using COMP_MVS_GRP_MISS. The means and corrected sums of squares for the original sample can then be calculated using UPDATE_MVS_GRP_MISS.

The means, variances and standard-deviations for the original sample can be computed by a final call to COMP_MVS_GRP_MISS with LAST=true.

This subroutine is adapted from:

(1) **Chan, T.F., Golub, G.H., and Leveque, R.J., 1979:** Updating formulae and a pairwise algorithm for computing sample variances. STAN-CS-79-773, November 1979.

6.22.34 subroutine update_mvs_grp (xmean_grp, xstd_grp, xn_grp, xmean_grp2, xstd_grp2, xn_grp2)

Purpose

UPDATE_MVS_GRP_MISS computes sample means and corrected sums of squares by groups for a sample, possibly containing missing values, given the means and corrected sums of squares for two subsamples as output by a call to COMP_MVS_GRP_MISS when LAST=false on the two subsamples separetely.

The sample means, variances and standard-deviations for the sample may be obtained by a call to COMP_MVS_GRP_MISS with LAST=true.

Arguments

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, the sample means for the groups computed on the first sample matrix.

On exit, the sample means for the groups computed on the combined sample matrix.

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, the sample corrected sums of squares for the groups computed on the first sample matrix.

On exit, the sample corrected sums of squares for the groups computed on the combined sample matrix.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(XMEAN_GRP,1)
- size(XSTD_GRP,2) = size(XMEAN_GRP,2)
- size(XSTD_GRP,3) = size(XMEAN_GRP,3).

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, the number of observations in each group in the first sample matrix.

On exit, the number of observations in each group in the combined sample matrix.

The shape of XSTD_GRP must verify:

- size(XN_GRP,1) = size(XMEAN_GRP,1)
- size(XN_GRP,2) = size(XMEAN_GRP,2)
- size(XN_GRP,3) = size(XMEAN_GRP,3).

XMEAN_GRP2 (INPUT) real(stnd), dimension(:,:,:) On entry, the sample means for the groups computed on the second sample matrix.

The shape of XMEAN_GRP2 must verify:

- size(XMEAN_GRP2,1) = size(XMEAN_GRP,1)
- size(XMEAN_GRP2,2) = size(XMEAN_GRP,2)
- size(XMEAN_GRP2,3) = size(XMEAN_GRP,3).

XSTD_GRP2 (INPUT) real(stnd), dimension(:,:,:) On entry, the sample corrected sums of squares for the groups computed on the second sample matrix.

The shape of XSTD_GRP2 must verify:

- size(XSTD_GRP2,1) = size(XMEAN_GRP,1)
- size(XSTD_GRP2,2) = size(XMEAN_GRP,2)
- size(XSTD_GRP2,3) = size(XMEAN_GRP,3).

XN_GRP2 (INPUT) real(stnd), dimension(:,:,:) On entry, the number of observations in each group in the second sample matrix.

The shape of XSTD_GRP must verify:

- size(XN_GRP2,1) = size(XMEAN_GRP,1)
- size(XN_GRP2,2) = size(XMEAN_GRP,2)
- size(XN_GRP2,3) = size(XMEAN_GRP,3).

Further Details

One possible application of this subroutine is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares computed for each subsample independently using COMP_MVS_GRP_MISS. The means and corrected sums of squares for the original sample can then be calculated using UPDATE_MVS_GRP_MISS.

The means, variances and and standard-deviations for the original sample can be computed by a final call to COMP_MVS_GRP_MISS with LAST=true.

This subroutine is adapted from:

(1) **Chan, T.F., Golub, G.H., and Leveque, R.J., 1979:** Updating formulae and a pairwise algorithm for computing sample variances. STAN-CS-79-773, November 1979.

6.22.35 subroutine update_mvs_grp (xmean_grp, xstd_grp, xn_grp, xmean_grp2, xstd_grp2, xn_grp2)

Purpose

UPDATE_MVS_GRP_MISS computes sample means and corrected sums of squares by groups for a sample, possibly containing missing values, given the means and corrected sums of squares for two subsamples as output by a call to COMP_MVS_GRP_MISS when LAST=false on the two subsamples separetely.

The sample means, variances and standard-deviations for the sample may be obtained by a call to COMP_MVS_GRP_MISS with LAST=true.

Arguments

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:,:) On entry, the sample means for the groups computed on the first sample array.

On exit, the sample means for the groups computed on the combined sample array.

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:,:) On entry, the sample corrected sums of squares for the groups computed on the first sample array.

On exit, the sample corrected sums of squares for the groups computed on the combined sample array.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(XMEAN_GRP,1)
- size(XSTD_GRP,2) = size(XMEAN_GRP,2)
- size(XSTD_GRP,3) = size(XMEAN_GRP,3)
- size(XSTD_GRP,4) = size(XMEAN_GRP,4).

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:,:) On entry, the number of observations in each group in the first sample array.

On exit, the number of observations in each group in the combined sample array.

The shape of XSTD_GRP must verify:

- size(XN_GRP,1) = size(XMEAN_GRP,1)
- size(XN_GRP,2) = size(XMEAN_GRP,2)
- size(XN_GRP,3) = size(XMEAN_GRP,3)
- size(XN_GRP,4) = size(XMEAN_GRP,4).

XMEAN_GRP2 (INPUT) real(stnd), dimension(:,:,:,:) On entry, the sample means for the groups computed on the second sample array.

The shape of XMEAN_GRP2 must verify:

- size(XMEAN_GRP2,1) = size(XMEAN_GRP,1)
- size(XMEAN_GRP2,2) = size(XMEAN_GRP,2)
- size(XMEAN_GRP2,3) = size(XMEAN_GRP,3)
- size(XMEAN_GRP2,4) = size(XMEAN_GRP,4).

XSTD_GRP2 (INPUT) real(stnd), dimension(:,:,:,:) On entry, the sample corrected sums of squares for the groups computed on the second sample array.

The shape of XSTD_GRP2 must verify:

- size(XSTD_GRP2,1) = size(XMEAN_GRP,1)
- size(XSTD_GRP2,2) = size(XMEAN_GRP,2)
- size(XSTD_GRP2,3) = size(XMEAN_GRP,3)
- size(XSTD_GRP2,4) = size(XMEAN_GRP,4).

XN_GRP2 (INPUT) real(stnd), dimension(:,:,:,:) On entry, the number of observations in each group in the second sample array.

The shape of XSTD_GRP must verify:

- size(XN_GRP2,1) = size(XMEAN_GRP,1)
- size(XN_GRP2,2) = size(XMEAN_GRP,2)
- size(XN_GRP2,3) = size(XMEAN_GRP,3).
- size(XN_GRP2,4) = size(XMEAN_GRP,4).

Further Details

One possible application of this subroutine is to parallel processing. If one has two or more processors available, the sample can be split up into smaller subsamples, and the means and corrected sums of squares computed for each subsample independently using COMP_MVS_GRP_MISS. The means and corrected sums of squares for the original sample can then be calculated using UPDATE_MVS_GRP_MISS.

The means, variances and and standard-deviations for the original sample can be computed by a final call to COMP_MVS_GRP_MISS with LAST=true.

This subroutine is adapted from:

(1) **Chan, T.F., Golub, G.H., and Leveque, R.J., 1979:** Updating formulae and a pairwise algorithm for computing sample variances. STAN-CS-79-773, November 1979.

6.22.36 `subroutine comp_anoma ( x, xmean, xstd )`

Purpose

COMP_ANOMA computes (standardized) anomalies from a data vector.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:) On entry, input subvector containing size(X) observations from the vector of data for which standardization is desired. If all the data are available at once, X can be the full data vector.

XMEAN (INPUT) real(stnd) On entry, XMEAN contains the mean value of the data vector.

XSTD (INPUT, OPTIONAL) real(stnd) On entry, if XSTD is present, XSTD contains the standard deviation of the data vector and the anomalies are standardized.

Further Details

It is assumed that the argument XSTD is greater than zero.

6.22.37 subroutine comp_anoma (x, xmean, xstd, dimvar)

Purpose

COMP_ANOMA computes (standardized) anomalies from a data matrix.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which standardization is desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data vector.

XMEAN (INPUT) real(stnd), dimension(:) On entry, XMEAN contains the mean values.

The size of XMEAN must verify: size(XMEAN) = size(X,DIMVAR).

XSTD (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if XSTD is present, XSTD contains the standard deviations and the anomalies are standardized.

The size of XSTD must verify: size(XSTD) = size(X,DIMVAR).

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

Further Details

It is assumed that elements of the array argument XSTD are greater than zero.

6.22.38 subroutine comp_anoma (x, xmean, xstd)

Purpose

COMP_ANOMA computes (standardized) anomalies from a data matrix.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, input subarray containing size(X,3) observations on size(X,1) by size(X,2) variables from the array of data for which standardization is desired. If all the data are available at once, X can be the full data array.

XMEAN (INPUT) real(stnd), dimension(:,:) On entry, XMEAN contains the mean values.

The shape of XMEAN must verify:

- size(XMEAN,1) = size(X,1)
- size(XMEAN,2) = size(X,2).

XSTD (INPUT, OPTIONAL) real(stnd), dimension(:,:) On entry, if XSTD is present, XSTD contains the standard deviations and the anomalies are standardized.

The shape of XSTD must verify:

- size(XSTD,1) = size(X,1)
- size(XSTD,2) = size(X,2).

Further Details

It is assumed that elements of the array argument XSTD are greater than zero.

6.22.39 subroutine comp_anoma_miss (x, xmiss, xmean, xstd)

Purpose

COMP_ANOMA_MISS computes (standardized) anomalies from a data vector possibly containing missing values.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:) On entry, input subvector containing size(X) observations from the vector of data for which standardization is desired. If all the data are available at once, X can be the full data vector.

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

XMEAN (INPUT) real(stnd) On entry, XMEAN contains the mean value of the data vector.

XSTD (INPUT, OPTIONAL) real(stnd) On entry, if XSTD is present, XSTD contains the standard deviation of the data vector and the anomalies are standardized.

Further Details

It is assumed that the argument XMEAN is not missing.

It is assumed that the argument XSTD is greater than zero and is not missing.

6.22.40 subroutine comp_anoma_miss (x, xmiss, xmean, xstd, dimvar)

Purpose

COMP_ANOMA_MISS computes (standardized) anomalies from a data matrix possibly containing missing values.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which standardization is desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data vector.

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

XMEAN (INPUT) real(stnd), dimension(:) On entry, XMEAN contains the mean values.

The size of XMEAN must verify: size(XMEAN) = size(X,DIMVAR).

XSTD (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if XSTD is present, XSTD contains the standard deviations and the anomalies are standardized.

The size of XSTD must verify: size(XSTD) = size(X,DIMVAR).

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

Further Details

It is assumed that elements of the array argument XMEAN are not missing.

It is assumed that elements of the array argument XSTD are greater than zero and are not missing.

6.22.41 `subroutine comp_anoma_miss ( x, xmiss, xmean, xstd )`

Purpose

COMP_ANOMA_MISS computes (standardized) anomalies from a data matrix possibly containing missing values.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, input subarray containing size(X,3) observations on size(X,1) by size(X,2) variables from the array of data for which standardization is desired. If all the data are available at once, X can be the full data array.

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

XMEAN (INPUT) real(stnd), dimension(:,:) On entry, XMEAN contains the mean values.

The shape of XMEAN must verify:

- size(XMEAN,1) = size(X,1)
- size(XMEAN,2) = size(X,2).

XSTD (INPUT, OPTIONAL) real(stnd), dimension(:,:) On entry, if XSTD is present, XSTD contains the standard deviations and the anomalies are standardized.

The shape of XSTD must verify:

- size(XSTD,1) = size(X,1)
- size(XSTD,2) = size(X,2).

Further Details

It is assumed that elements of the array argument XMEAN are not missing.

It is assumed that elements of the array argument XSTD are greater than zero and are not missing.

6.22.42 subroutine comp_anoma_grp (x, ngrp, ind, xmean_grp, xstd_grp)

Purpose

COMP_ANOMA_GRP computes (standardized) anomalies by groups from a data vector.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:) On entry, input subvector containing size(X) observations from the vector of data for which standardization by groups is desired. If all the data are available at once, X can be the full data vector.

NGRP (INPUT) integer(i4b) On entry, the number of groups.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group and this observation is not standardized.

The size of IND must verify: size(IND) = size(X) .

XMEAN_GRP (INPUT) real(stnd), dimension(:) On entry, XMEAN_GRP contains the mean values for the NGRP groups of observations in the data vector.

The size of XMEAN_GRP must verify: size(XMEAN_GRP) = NGRP .

XSTD_GRP (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if XSTD_GRP is present, XSTD_GRP contains the standard deviations for the NGRP groups of observations in the data vector and the observations are standardized.

The size of XSTD_GRP must verify: size(XSTD_GRP) = NGRP .

Further Details

It is assumed that elements of the array argument XSTD_GRP are greater than zero.

6.22.43 subroutine comp_anoma_grp (x, ngrp, ind, xmean_grp, xstd_grp, dimvar)

Purpose

COMP_ANOMA_GRP computes (standardized) anomalies by groups from a data matrix.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which standardization by groups is desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data vector.

NGRP (INPUT) integer(i4b) On entry, the number of groups.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X,3-DIMVAR) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group and this observation is not standardized.

The size of IND must verify: size(IND) = size(X,3-DIMVAR) .

XMEAN_GRP (INPUT) real(stnd), dimension(:,:) On entry, XMEAN_GRP contains the mean values for the NGRP groups of observations in the data matrix.

The shape of XMEAN_GRP must verify:

- size(XMEAN_GRP,1) = size(X,DIMVAR)
- size(XMEAN_GRP,2) = NGRP.

XSTD_GRP (INPUT, OPTIONAL) real(stnd), dimension(:,:) On entry, if XSTD_GRP is present, XSTD_GRP contains the standard deviations for the NGRP groups of observations in the data matrix and the observations are standardized.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(X,DIMVAR)
- size(XSTD_GRP,2) = NGRP.

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

Further Details

It is assumed that elements of the array argument XSTD_GRP are greater than zero.

6.22.44 subroutine comp_anoma_grp (x, ngrp, ind, xmean_grp, xstd_grp)

Purpose

COMP_ANOMA_GRP computes (standardized) anomalies by groups from a data matrix possibly containing missing values.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, input subarray containing size(X,3) observations on size(X,1) by size(X,2) variables from the array of data for which standardization by groups is desired. If all the data are available at once, X can be the full data array.

NGRP (INPUT) integer(i4b) On entry, the number of groups.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X,3) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group and this observation is not standardized.

The size of IND must verify: size(IND) = size(X,3) .

XMEAN_GRP (INPUT) real(stnd), dimension(:,:,:) On entry, XMEAN_GRP contains the mean values for the NGRP groups of observations in the data array.

The shape of XMEAN_GRP must verify:

- size(XMEAN_GRP,1) = size(X,1)
- size(XMEAN_GRP,2) = size(X,2)
- size(XMEAN_GRP,3) = NGRP.

XSTD_GRP (INPUT, OPTIONAL) real(stnd), dimension(:,:,:) On entry, if XSTD_GRP is present, XSTD_GRP contains the standard deviations for the NGRP groups of observations in the data array and the observations are standardized.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(X,1)
- size(XSTD_GRP,2) = size(X,2)
- size(XSTD_GRP,3) = NGRP.

Further Details

It is assumed that elements of the array argument XSTD_GRP are greater than zero.

6.22.45 subroutine comp_anoma_grp_miss (x, ngrp, ind, xmiss, xmean_grp, xstd_grp)

Purpose

COMP_ANOMA_GRP_MISS computes (standardized) anomalies by groups from a data vector possibly containing missing values.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:) On entry, input subvector containing size(X) observations from the vector of data for which standardization by groups is desired. If all the data are available at once, X can be the full data vector.

NGRP (INPUT) integer(i4b) On entry, the number of groups.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group and this observation is not standardized.

The size of IND must verify: size(IND) = size(X) .

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

XMEAN_GRP (INPUT) real(stnd), dimension(:) On entry, XMEAN_GRP contains the mean values for the NGRP groups of observations in the data vector.

The size of XMEAN_GRP must verify: size(XMEAN_GRP) = NGRP .

XSTD_GRP (INPUT, OPTIONAL) real(stnd), dimension(:) On entry, if XSTD_GRP is present, XSTD_GRP contains the standard deviations for the NGRP groups of observations in the data vector and the observations are standardized.

The size of XSTD_GRP must verify: size(XSTD_GRP) = NGRP .

Further Details

It is assumed that elements of the array argument XMEAN_GRP are not missing.

It is assumed that elements of the array argument XSTD_GRP are greater than zero and are not missing.

6.22.46 subroutine comp_anoma_grp_miss (x, ngrp, ind, xmiss, xmean_grp, xstd_grp, dimvar)

Purpose

COMP_ANOMA_GRP_MISS computes (standardized) anomalies by groups from a data matrix possibly containing missing values.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which standardization by groups is desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data vector.

NGRP (INPUT) integer(i4b) On entry, the number of groups.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X,3-DIMVAR) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group and this observation is not standardized.

The size of IND must verify: size(IND) = size(X,3-DIMVAR) .

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

XMEAN_GRP (INPUT) real(stnd), dimension(:,:) On entry, XMEAN_GRP contains the mean values for the NGRP groups of observations in the data matrix.

The shape of XMEAN_GRP must verify:

- size(XMEAN_GRP,1) = size(X,DIMVAR)
- size(XMEAN_GRP,2) = NGRP.

XSTD_GRP (INPUT, OPTIONAL) real(stnd), dimension(:,:) On entry, if XSTD_GRP is present, XSTD_GRP contains the standard deviations for the NGRP groups of observations in the data matrix and the observations are standardized.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(X,DIMVAR)
- size(XSTD_GRP,2) = NGRP.

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

Further Details

It is assumed that elements of the array argument XMEAN_GRP are not missing.

It is assumed that elements of the array argument XSTD_GRP are greater than zero and are not missing.

6.22.47 subroutine comp_anoma_grp_miss (x, ngrp, ind, xmiss, xmean_grp, xstd_grp)

Purpose

COMP_ANOMA_GRP_MISS computes (standardized) anomalies by groups from a data matrix possibly containing missing values.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, input subarray containing size(X,3) observations on size(X,1) by size(X,2) variables from the array of data for which standardization by groups is desired. If all the data are available at once, X can be the full data array.

NGRP (INPUT) integer(i4b) On entry, the number of groups.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X,3) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group and this observation is not standardized.

The size of IND must verify: size(IND) = size(X,3) .

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

XMEAN_GRP (INPUT) real(stnd), dimension(:,:,:) On entry, XMEAN_GRP contains the mean values for the NGRP groups of observations in the data array.

The shape of XMEAN_GRP must verify:

- size(XMEAN_GRP,1) = size(X,1)
- size(XMEAN_GRP,2) = size(X,2)
- size(XMEAN_GRP,3) = NGRP.

XSTD_GRP (INPUT, OPTIONAL) real(stnd), dimension(:,:,:) On entry, if XSTD_GRP is present, XSTD_GRP contains the standard deviations for the NGRP groups of observations in the data array and the observations are standardized.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(X,1)
- size(XSTD_GRP,2) = size(X,2)
- size(XSTD_GRP,3) = NGRP.

Further Details

It is assumed that elements of the array argument XMEAN_GRP are not missing.

It is assumed that elements of the array argument XSTD_GRP are greater than zero and are not missing.

6.22.48 subroutine comp_composite (x, first, last, ngrp, ind, xmean, xstd, xn, xmean_grp, xstd_grp, xn_grp, xcomp, u, prob, utest)

Purpose

COMP_COMPOSITE computes a composite analysis from a data vector.

Arguments

X (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X) observations from the vector of data for which a composite analysis is desired. If all the data are available at once, X can be the full data vector.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subvector is the first subvector of the data vector.
- FIRST = false the current subvector is not the first subvector of the data vector.

LAST (INPUT) logical(lgl) On entry, if:

- **LAST = true the current subvector is the last subvector** of the data vector.
- **LAST = false the current subvector is not the last subvector** of the data vector.

NGRP (INPUT) integer(i4b) On entry, the number of groups in the analysis.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group in the analysis.

The size of IND must verify: size(IND) = size(X) .

XMEAN (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_COMPOSITE (e.g. when FIRST=true), XMEAN contains the mean value from previous calls to COMP_COMPOSITE. XMEAN should not be changed between calls to COMP_COMPOSITE.

On exit, when LAST=true, XMEAN contains the mean value of the data vector.

XSTD (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_COMPOSITE (e.g. when FIRST=true), XSTD contains adjusted sum of squares from previous calls to COMP_COMPOSITE. XSTD should not be changed between calls to COMP_COMPOSITE.

On exit, when LAST=true, XSTD contains the standard deviation of the data vector.

XN (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_COMPOSITE (e.g. when FIRST=true), XN contains count of observations from previous calls to COMP_COMPOSITE. XN should not be changed between calls to COMP_COMPOSITE.

On exit, when LAST=true, XN contains the number of observations in the data vector.

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_COMPOSITE (e.g. when FIRST=true), XMEAN_GRP contains the mean values for the NGRP groups from previous calls to COMP_COMPOSITE. XMEAN_GRP should not be changed between calls to COMP_COMPOSITE.

On exit, when LAST=true, XMEAN_GRP contains the mean values for the NGRP groups of observations in the data vector.

The size of XMEAN_GRP must verify: size(XMEAN_GRP) = NGRP .

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_COMPOSITE (e.g. when FIRST=true), XSTD_GRP contains adjusted sums of squares for the NGRP groups from previous calls to COMP_COMPOSITE. XSTD_GRP should not be changed between calls to COMP_COMPOSITE.

On exit, when LAST=true, XSTD_GRP contains the standard deviations for the NGRP groups of observations in the data vector.

The size of XSTD_GRP must verify: size(XSTD_GRP) = NGRP .

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_COMPOSITE (e.g. when FIRST=true), XN_GRP contains counts of observations for the NGRP groups from previous calls to COMP_COMPOSITE. XN_GRP should not be changed between calls to COMP_COMPOSITE.

On exit, when LAST=true, XN_GRP contains the numbers of observations in the NGRP groups of observations in the data vector.

The size of XN_GRP must verify: size(XN_GRP) = NGRP .

XCOMP (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the standardized profiles of the NGRP groups of observations in the data vector.

The size of XCOMP must verify: size(XCOMP) = NGRP .

U (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the U statistics for the NGRP groups of observations in the data vector.

The size of U must verify: size(U) = NGRP .

PROB (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the significance probabilities of the U statistics under the null hypothesis that the groups have been formed by uniform random sampling without duplication in the set of all the observations.

The size of PROB must verify: size(PROB) = NGRP .

UTEST (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, a probability. UTEST is the sum of the areas (equal) in both tails of the normal-distribution. UTEST must verify: 0. < P < 1.

On exit, the two-tail quantile of the normal-distribution, that is a value X such that the probability of the absolute value of U being greater than X is UTEST under the null hypothesis that the groups have been formed by uniform random sampling without duplication in the set of all the observations.

Further Details

The subroutine computes all the statistics with only one pass through the data.

If fewer than one observation is present, the statistics are set to Nan code.

The optional parameters need only to be specified when LAST=true.

For more details, on the statistics and tests computed by this subroutine, see:

(1) **Terray, P., Delecluse, P., Labattu, S., Terray, L., 2003:** Sea Surface Temperature associations with the Late Indian Summer Monsoon. Climate Dynamics, vol. 21, 593-618.

6.22.49 subroutine comp_composite (x, first, last, ngrp, ind, xmean, xstd, xn, xmean_grp, xstd_grp, xn_grp, dimvar, xcomp, u, prob, utest)

Purpose

COMP_COMPOSITE computes a composite analysis from a data matrix.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which a composite analysis is desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrix is the first submatrix of the data matrix.
- FIRST = false the current submatrix is not the first submatrix of the data matrix.

LAST (INPUT) logical(lgl) On entry, if:

- **LAST = true the current submatrix is the last submatrix** of the data matrix.
- **LAST = false the current submatrix is not the last submatrix** of the data matrix.

NGRP (INPUT) integer(i4b) On entry, the number of groups in the analysis.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X,3-DIMVAR) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group in the analysis.

The size of IND must verify: size(IND) = size(X,3-DIMVAR) .

XMEAN (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_COMPOSITE (e.g. when FIRST=true), XMEAN contains the mean values from previous calls to COMP_COMPOSITE. XMEAN should not be changed between calls to COMP_COMPOSITE.

On exit, when LAST=true, XMEAN contains the mean values of the data matrix.

The size of XMEAN must verify: size(XMEAN) = size(X,DIMVAR).

XSTD (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_COMPOSITE (e.g. when FIRST=true), XSTD contains adjusted sums of squares from previous calls to COMP_COMPOSITE. XSTD should not be changed between calls to COMP_COMPOSITE.

On exit, when LAST=true, XSTD contains the standard deviations of the data matrix.

The size of XSTD must verify: size(XSTD) = size(X,DIMVAR).

XN (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_COMPOSITE (e.g. when FIRST=true), XN contains count of observations from previous calls to COMP_COMPOSITE. XN should not be changed between calls to COMP_COMPOSITE.

On exit, XN contains the number of observations in the data matrix.

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_COMPOSITE (e.g. when FIRST=true), XMEAN_GRP contains the mean values for the NGRP groups from previous calls to COMP_COMPOSITE. XMEAN_GRP should not be changed between calls to COMP_COMPOSITE.

On exit, when LAST=true, XMEAN_GRP contains the mean values for the NGRP groups of observations on all the variables in the data matrix.

The shape of XMEAN_GRP must verify:

- size(XMEAN_GRP,1) = size(X,DIMVAR)
- size(XMEAN_GRP,2) = NGRP.

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_COMPOSITE (e.g. when FIRST=true), XSTD_GRP contains adjusted sums of squares for the NGRP groups from previous calls to COMP_COMPOSITE. XSTD_GRP should not be changed between calls to COMP_COMPOSITE.

On exit, when LAST=true, XSTD_GRP contains the standard deviations for the NGRP groups of observations on all the variables in the data matrix.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(X,DIMVAR)
- size(XSTD_GRP,2) = NGRP.

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_COMPOSITE (e.g. when FIRST=true), XN_GRP contains counts of observations for the NGRP groups from previous calls to COMP_COMPOSITE. XN_GRP should not be changed between calls to COMP_COMPOSITE.

On exit, when LAST=true, XN_GRP contains the numbers of observations in the NGRP groups for all the variables in the data matrix.

The size of XN_GRP must verify: size(XN_GRP) = NGRP .

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

XCOMP (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, the standardized profiles of the NGRP groups of observations in the data matrix.

The shape of XCOMP must verify:

- size(XCOMP,1) = size(X,DIMVAR)
- size(XCOMP,2) = NGRP.

U (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, the U statistics for the NGRP groups and all the variables in the data matrix.

The shape of U must verify:

- size(U,1) = size(X,DIMVAR)
- size(U,2) = NGRP.

PROB (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, the significance probabilities of the U statistics under the null hypothesis that the groups have been formed by uniform random sampling without duplication in the set of all the observations.

The shape of PROB must verify:

- size(PROB,1) = size(X,DIMVAR)
- size(PROB,2) = NGRP.

UTEST (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, a probability. UTEST is the sum of the areas (equal) in both tails of the normal-distribution. UTEST must verify: 0. < P < 1.

On exit, the two-tail quantile of the normal-distribution, that is a value X such that the probability of the absolute value of U being greater than X is UTEST under the null hypothesis that the groups have been formed by uniform random sampling without duplication in the set of all the observations.

Further Details

The subroutine computes all the statistics with only one pass through the data.

If fewer than one observation is present, the statistics are set to Nan code.

The optional parameters, except DIMVAR, need only to be specified when LAST=true.

For more details, on the statistics and tests computed by this subroutine, see:

(1) **Terray, P., Delecluse, P., Labattu, S., Terray, L., 2003:** Sea Surface Temperature associations with the Late Indian Summer Monsoon. Climate Dynamics, vol. 21, 593-618.

6.22.50 subroutine comp_composite (x, first, last, ngrp, ind, xmean, xstd, xn, xmean_grp, xstd_grp, xn_grp, xcomp, u, prob, utest)

Purpose

COMP_COMPOSITE computes a composite analysis from a data tridimensional array.

Arguments

X (INPUT) real(stnd), dimension(:,:,:) On entry, input subarray containing size(X,3) observations on size(X,1) by size(X,2) variables from the array of data for which a composite analysis is desired. If all the data are available at once, X can be the full data array.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subarray is the first subarray of the data array.
- FIRST = false the current subarray is not the first subarray of the data array.

LAST (INPUT) logical(lgl) On entry, if:

- **LAST = true the current subarray is the last subarray** of the data array.
- **LAST = false the current subarray is not the last subarray** of the data array.

NGRP (INPUT) integer(i4b) On entry, the number of groups in the analysis.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X,3) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group in the analysis.

The size of IND must verify: size(IND) = size(X,3) .

XMEAN (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_COMPOSITE (e.g. when FIRST=true), XMEAN contains the mean values from previous calls to COMP_COMPOSITE. XMEAN should not be changed between calls to COMP_COMPOSITE.

On exit, when LAST=true, XMEAN contains the mean values of the data array.

The shape of XMEAN must verify:

- size(XMEAN,1) = size(X,1)
- size(XMEAN,2) = size(X,2).

XSTD (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_COMPOSITE (e.g. when FIRST=true), XSTD contains adjusted sums of squares from previous calls to COMP_COMPOSITE. XSTD should not be changed between calls to COMP_COMPOSITE.

On exit, when LAST=true, XSTD contains the standard deviations of the data array.

The shape of XSTD must verify:

- size(XSTD,1) = size(X,1)
- size(XSTD,2) = size(X,2).

XN (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_COMPOSITE (e.g. when FIRST=true), XN contains counts of observations from previous calls to COMP_COMPOSITE. XN should not be changed between calls to COMP_COMPOSITE.

On exit, when LAST=true, XN contains the number of observations in the data array.

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, after the first call to COMP_COMPOSITE (e.g. when FIRST=true), XMEAN_GRP contains the mean values for the NGRP groups from previous calls to COMP_COMPOSITE. XMEAN_GRP should not be changed between calls to COMP_COMPOSITE.

On exit, when LAST=true, XMEAN_GRP contains the mean values for the NGRP groups of observations on all the variables in the data array.

The shape of XMEAN_GRP must verify:

- size(XMEAN_GRP,1) = size(X,1)
- size(XMEAN_GRP,2) = size(X,2)
- size(XMEAN_GRP,3) = NGRP.

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, after the first call to COMP_COMPOSITE (e.g. when FIRST=true), XSTD_GRP contains adjusted sums of squares for the NGRP groups from previous calls to COMP_COMPOSITE. XSTD_GRP should not be changed between calls to COMP_COMPOSITE.

On exit, when LAST=true, XSTD_GRP contains the standard deviations for the NGRP groups of observations on all the variables in the data array.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(X,1)
- size(XSTD_GRP,2) = size(X,2)
- size(XSTD_GRP,3) = NGRP.

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_COMPOSITE (e.g. when FIRST=true), XN_GRP contains counts of observations for the NGRP groups from previous calls to COMP_COMPOSITE. XN_GRP should not be changed between calls to COMP_COMPOSITE.

On exit, when LAST=true, XN_GRP contains the numbers of observations in the NGRP groups for all the variables in the data array.

The size of XN_GRP must verify: size(XN_GRP) = NGRP .

XCOMP (OUTPUT, OPTIONAL) real(stnd), dimension(:,:,:) On exit, the standardized profiles of the NGRP groups of observations in the data array.

The shape of XCOMP must verify:

- size(XCOMP,1) = size(X,1)
- size(XCOMP,2) = size(X,2)
- size(XCOMP,3) = NGRP.

U (OUTPUT, OPTIONAL) real(stnd), dimension(:,:,:) On exit, the U statistics for the NGRP groups and all the variables in the data array.

The shape of U must verify:

- size(U,1) = size(X,1)
- size(U,2) = size(X,2)
- size(U,3) = NGRP.

PROB (OUTPUT, OPTIONAL) real(stnd), dimension(:,:,:) On exit, the significance probabilities of the U statistics under the null hypothesis that the groups have been formed by uniform random sampling without duplication in the set of all the observations.

The shape of PROB must verify:

- size(PROB,1) = size(X,1)
- size(PROB,2) = size(X,2)
- size(PROB,3) = NGRP.

UTEST (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, a probability. UTEST is the sum of the areas (equal) in both tails of the normal-distribution. UTEST must verify: 0. < P < 1.

On exit, the two-tail quantile of the normal-distribution, that is a value X such that the probability of the absolute value of U being greater than X is UTEST under the null hypothesis that the groups have been formed by uniform random sampling without duplication in the set of all the observations.

Further Details

The subroutine computes all the statistics with only one pass through the data.

If fewer than one observation is present, the statistics are set to Nan code.

The optional parameters need only to be specified when LAST=true.

For more details, on the statistics and tests computed by this subroutine, see:

(1) **Terray, P., Delecluse, P., Labattu, S., Terray, L., 2003:** Sea Surface Temperature associations with the Late Indian Summer Monsoon. Climate Dynamics, vol. 21, 593-618.

6.22.51 subroutine comp_composite (x, first, last, ngrp, ind, xmean, xstd, xn, xmean_grp, xstd_grp, xn_grp, xmiss, xcomp, u, prob, utest)

Purpose

COMP_COMPOSITE_MISS computes a composite analysis from a data vector possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:) On entry, input subvector containing size(X) observations from the vector of data for which a composite analysis is desired. If all the data are available at once, X can be the full data vector.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subvector is the first subvector of the data vector.
- FIRST = false the current subvector is not the first subvector of the data vector.

LAST (INPUT) logical(lgl) On entry, if:

- **LAST = true the current subvector is the last subvector** of the data vector.
- **LAST = false the current subvector is not the last subvector** of the data vector.

NGRP (INPUT) integer(i4b) On entry, the number of groups in the analysis.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group in the analysis.

The size of IND must verify: size(IND) = size(X) .

XMEAN (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_COMPOSITE_MISS (e.g. when FIRST=true), XMEAN contains the mean value from previous calls to COMP_COMPOSITE_MISS. XMEAN should not be changed between calls to COMP_COMPOSITE_MISS.

On exit, when LAST=true, XMEAN contains the mean value of the data vector.

XSTD (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_COMPOSITE_MISS (e.g. when FIRST=true), XSTD contains adjusted sum of squares from previous calls to COMP_COMPOSITE_MISS. XSTD should not be changed between calls to COMP_COMPOSITE_MISS.

On exit, when LAST=true, XSTD contains the standard deviation of the data vector.

XN (INPUT/OUTPUT) real(stnd) On entry, after the first call to COMP_COMPOSITE_MISS (e.g. when FIRST=true), XN contains count of non-missing observations from previous calls to COMP_COMPOSITE_MISS. XN should not be changed between calls to COMP_COMPOSITE_MISS.

On exit, when LAST=true, XN contains the number of observations in the data vector.

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_COMPOSITE_MISS (e.g. when FIRST=true), XMEAN_GRP contains the mean values for the NGRP groups from previous calls to COMP_COMPOSITE_MISS. XMEAN_GRP should not be changed between calls to COMP_COMPOSITE_MISS.

On exit, when LAST=true, XMEAN_GRP contains the mean values for the NGRP groups of observations in the data vector.

The size of XMEAN_GRP must verify: size(XMEAN_GRP) = NGRP .

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_COMPOSITE_MISS (e.g. when FIRST=true), XSTD_GRP contains adjusted sums of squares for the NGRP groups from previous calls to COMP_COMPOSITE_MISS. XSTD_GRP should not be changed between calls to COMP_COMPOSITE_MISS.

On exit, when LAST=true, XSTD_GRP contains the standard deviations for the NGRP groups of observations in the data vector.

The size of XSTD_GRP must verify: size(XSTD_GRP) = NGRP .

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_COMPOSITE_MISS (e.g. when FIRST=true), XN_GRP contains counts of non-missing observations for the NGRP groups from previous calls to COMP_COMPOSITE_MISS. XN_GRP should not be changed between calls to COMP_COMPOSITE_MISS.

On exit, XN_GRP contains the number of non-missing observations in the NGRP groups of observations in the data vector.

The size of XN_GRP must verify: size(XN_GRP) = NGRP .

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

XCOMP (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the standardized profiles of the NGRP groups.

The size of XCOMP must verify: size(XCOMP) = NGRP .

U (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the U statistics for the NGRP groups of observations in the data vector.

The size of U must verify: size(U) = NGRP .

PROB (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the significance probabilities of the U statistics under the null hypothesis that the groups have been formed by uniform random sampling without duplication in the set of all the observations.

The size of PROB must verify: size(PROB) = NGRP .

UTEST (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, a probability. UTEST is the sum of the areas (equal) in both tails of the normal-distribution. UTEST must verify: 0. < P < 1.

On exit, the two-tail quantile of the normal-distribution, that is a value X such that the probability of the absolute value of U being greater than X is UTEST under the null hypothesis that the groups have been formed by uniform random sampling without duplication in the set of all the observations.

Further Details

The subroutine computes all the statistics with only one pass through the data.

If fewer than one valid observation were present, the statistics are set to Nan code.

If fewer than one valid observation were present only for some groups of observations, the pertinent statistics are set to missing (XMISS value).

The optional parameters need only to be specified when LAST=true.

For more details, on the statistics and tests computed by this subroutine, see:

(1) **Terray, P., Delecluse, P., Labattu, S., Terray, L., 2003:** Sea Surface Temperature associations with the Late Indian Summer Monsoon. Climate Dynamics, vol. 21, 593-618.

6.22.52 subroutine comp_composite (x, first, last, ngrp, ind, xmean, xstd, xn, xmean_grp, xstd_grp, xn_grp, xmiss, dimvar, xcomp, u, prob, utest)

Purpose

COMP_COMPOSITE_MISS computes a composite analysis from a data matrix possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, input submatrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables from the matrix of data for which a composite analysis is desired. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details. If all the data are available at once, X can be the full data matrix.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current submatrix is the first submatrix of the data matrix.
- FIRST = false the current submatrix is not the first submatrix of the data matrix.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current submatrix is the last submatrix of the data matrix.
- LAST = false the current submatrix is not the last submatrix of the data matrix.

NGRP (INPUT) integer(i4b) On entry, the number of groups in the analysis.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X,3-DIMVAR) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group in the analysis.

The size of IND must verify: size(IND) = size(X,3-DIMVAR) .

XMEAN (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_COMPOSITE_MISS (e.g. when FIRST=true), XMEAN contains the mean values from previous calls to COMP_COMPOSITE_MISS. XMEAN should not be changed between calls to COMP_COMPOSITE_MISS.

On exit, when LAST=true, XMEAN contains the mean values of the data matrix.

The size of XMEAN must verify: size(XMEAN) = size(X,DIMVAR).

XSTD (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_COMPOSITE_MISS (e.g. when FIRST=true), XSTD contains adjusted sums of squares from previous calls to COMP_COMPOSITE_MISS. XSTD should not be changed between calls to COMP_COMPOSITE_MISS.

On exit, when LAST=true, XSTD contains the standard deviations of the data matrix.

The size of XSTD must verify: size(XSTD) = size(X,DIMVAR).

XN (INPUT/OUTPUT) real(stnd), dimension(:) On entry, after the first call to COMP_COMPOSITE_MISS (e.g. when FIRST=true), XN contains counts of non-missing observations from previous calls to COMP_COMPOSITE_MISS. XN should not be changed between calls to COMP_COMPOSITE_MISS.

On exit, XN contains the numbers of non-missing observations for the variables in the data matrix.

The size of XN must verify: size(XN) = size(X,DIMVAR).

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_COMPOSITE_MISS (e.g. when FIRST=true), XMEAN_GRP contains the mean values for the NGRP groups from previous calls to COMP_COMPOSITE_MISS. XMEAN_GRP should not be changed between calls to COMP_COMPOSITE_MISS.

On exit, when LAST=true, XMEAN_GRP contains the mean values for the NGRP groups of observations on all the variables in the data matrix.

The shape of XMEAN_GRP must verify:

- size(XMEAN_GRP,1) = size(X,DIMVAR)
- size(XMEAN_GRP,2) = NGRP.

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_COMPOSITE_MISS (e.g. when FIRST=true), XSTD_GRP contains adjusted sums of squares for the NGRP groups from previous calls to COMP_COMPOSITE_MISS. XSTD_GRP should not be changed between calls to COMP_COMPOSITE_MISS.

On exit, when LAST=true, XSTD_GRP contains the standard deviations for the NGRP groups of observations on all the variables in the data matrix.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(X,DIMVAR)

- size(XSTD_GRP,2) = NGRP.

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_COMPOSITE_MISS (e.g. when FIRST=true), XN_GRP contains counts of non-missing observations for the NGRP groups from previous calls to COMP_COMPOSITE_MISS. XN_GRP should not be changed between calls to COMP_COMPOSITE_MISS.

On exit, XN_GRP contains the numbers of non-missing observations in the NGRP groups for all the variables in the data matrix.

The shape of XN_GRP must verify:

- size(XN_GRP,1) = size(X,DIMVAR)
- size(XN_GRP,2) = NGRP.

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows:

- DIMVAR = 1, the input submatrix X contains size(X,2) observations on size(X,1) variables.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables.

The default is DIMVAR = 1.

XCOMP (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, the standardized profiles of the NGRP groups of observations in the data matrix.

The shape of XCOMP must verify:

- size(XCOMP,1) = size(X,DIMVAR)
- size(XCOMP,2) = NGRP.

U (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, the U statistics for the NGRP groups and all the variables in the data matrix.

The shape of U must verify:

- size(U,1) = size(X,DIMVAR)
- size(U,2) = NGRP.

PROB (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, the significance probabilities of the U statistics under the null hypothesis that the groups have been formed by uniform random sampling without duplication in the set of all the observations.

The shape of PROB must verify:

- size(PROB,1) = size(X,DIMVAR)
- size(PROB,2) = NGRP.

UTEST (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, a probability. UTEST is the sum of the areas (equal) in both tails of the normal-distribution. UTEST must verify: 0. < P < 1.

On exit, the two-tail quantile of the normal-distribution, that is a value X such that the probability of the absolute value of U being greater than X is UTEST under the null hypothesis that the groups have been formed by uniform random sampling without duplication in the set of all the observations.

Further Details

The subroutine computes all the statistics with only one pass through the data.

If fewer than one valid observation were present for all variables, the statistics are set to Nan code.

If fewer than one valid observation were present only for some variables and/or groups of observations, the pertinent statistics are set to missing (XMISS value).

The optional parameters, except DIMVAR, need only to be specified when LAST=true.

For more details, on the statistics and tests computed by this subroutine, see:

(1) **Terray, P., Delecluse, P., Labattu, S., Terray, L., 2003:** Sea Surface Temperature associations with the Late Indian Summer Monsoon. Climate Dynamics, vol. 21, 593-618.

6.22.53 `subroutine comp_composite ( x, first, last, ngrp, ind, xmean, xstd, xn, xmean_grp, xstd_grp, xn_grp, xmiss, xcomp, u, prob, utest )`

Purpose

COMP_COMPOSITE_MISS computes a composite analysis from a data tridimensional array possibly containing missing values.

Arguments

X (INPUT) real(stnd), dimension(:,:,:) On entry, input subarray containing size(X,3) observations on size(X,1) by size(X,2) variables from the array of data for which a composite analysis is desired. If all the data are available at once, X can be the full data array.

FIRST (INPUT) logical(lgl) On entry, if:

- FIRST = true the current subarray is the first subarray of the data array.
- FIRST = false the current subarray is not the first subarray of the data array.

LAST (INPUT) logical(lgl) On entry, if:

- LAST = true the current subarray is the last subarray of the data array.
- LAST = false the current subarray is not the last subarray of the data array.

NGRP (INPUT) integer(i4b) On entry, the number of groups in the analysis.

IND (INPUT) integer(i4b), dimension(:) On entry, input subvector containing size(X,3) observations which is used to classify the observations into the NGRP groups. A value outside the interval 1:NGRP means that the current observation does not belong to any group in the analysis.

The size of IND must verify: size(IND) = size(X,3) .

XMEAN (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_COMPOSITE_MISS (e.g. when FIRST=true), XMEAN contains the mean values from previous calls to COMP_COMPOSITE_MISS. XMEAN should not be changed between calls to COMP_COMPOSITE_MISS.

On exit, when LAST=true, XMEAN contains the mean values of the data array.

The shape of XMEAN must verify:

- size(XMEAN,1) = size(X,1)

- size(XMEAN,2) = size(X,2).

XSTD (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_COMPOSITE_MISS (e.g. when FIRST=true), XSTD contains adjusted sums of squares from previous calls to COMP_COMPOSITE_MISS. XSTD should not be changed between calls to COMP_COMPOSITE_MISS.

On exit, when LAST=true, XSTD contains the standard deviations of the data array.

The shape of XSTD must verify:

- size(XSTD,1) = size(X,1)
- size(XSTD,2) = size(X,2).

XN (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, after the first call to COMP_COMPOSITE_MISS (e.g. when FIRST=true), XN contains counts of non-missing observations from previous calls to COMP_COMPOSITE_MISS. XN should not be changed between calls to COMP_COMPOSITE_MISS.

On exit, when LAST=true, XN contains the numbers of non-missing observations for the variables in the data array.

The shape of XN must verify:

- size(XN,1) = size(X,1)
- size(XN,2) = size(X,2).

XMEAN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, after the first call to COMP_COMPOSITE_MISS (e.g. when FIRST=true), XMEAN_GRP contains the mean values for the NGRP groups from previous calls to COMP_COMPOSITE_MISS. XMEAN_GRP should not be changed between calls to COMP_COMPOSITE_MISS.

On exit, when LAST=true, XMEAN_GRP contains the mean values for the NGRP groups of observations on all the variables in the data array.

The shape of XMEAN_GRP must verify:

- size(XMEAN_GRP,1) = size(X,1)
- size(XMEAN_GRP,2) = size(X,2)
- size(XMEAN_GRP,3) = NGRP.

XSTD_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, after the first call to COMP_COMPOSITE_MISS (e.g. when FIRST=true), XSTD_GRP contains adjusted sums of squares for the NGRP groups from previous calls to COMP_COMPOSITE_MISS. XSTD_GRP should not be changed between calls to COMP_COMPOSITE_MISS.

On exit, when LAST=true, XSTD_GRP contains the standard deviations for the NGRP groups of observations on all the variables in the data array.

The shape of XSTD_GRP must verify:

- size(XSTD_GRP,1) = size(X,1)
- size(XSTD_GRP,2) = size(X,2)
- size(XSTD_GRP,3) = NGRP.

XN_GRP (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, after the first call to COMP_COMPOSITE_MISS (e.g. when FIRST=true), XN_GRP contains counts of non-missing observations for the NGRP groups from previous calls to COMP_COMPOSITE_MISS. XN_GRP should not be changed between calls to COMP_COMPOSITE_MISS.

On exit, when LAST=true, XN_GRP contains the numbers of non-missing observations in the NGRP groups for all the variables in the data array.

The shape of XN_GRP must verify:

- size(XN_GRP,1) = size(X,1)
- size(XN_GRP,2) = size(X,2)
- size(XN_GRP,3) = NGRP.

XMISS (INPUT) real(stnd) On entry, the missing value indicator. Any value in X which is equal to XMISS is assumed to be missing.

XCOMP (OUTPUT, OPTIONAL) real(stnd), dimension(:,:,:) On exit, the standardized profiles of the NGRP groups of observations in the data array.

The shape of XCOMP must verify:

- size(XCOMP,1) = size(X,1)
- size(XCOMP,2) = size(X,2)
- size(XCOMP,3) = NGRP.

U (OUTPUT, OPTIONAL) real(stnd), dimension(:,:,:) On exit, the U statistics for the NGRP groups and all the variables in the data array.

The shape of U must verify:

- size(U,1) = size(X,1)
- size(U,2) = size(X,2)
- size(U,3) = NGRP.

PROB (OUTPUT, OPTIONAL) real(stnd), dimension(:,:,:) On exit, the significance probabilities of the U statistics under the null hypothesis that the groups have been formed by uniform random sampling without duplication in the set of all the observations.

The shape of PROB must verify:

- size(PROB,1) = size(X,1)
- size(PROB,2) = size(X,2)
- size(PROB,3) = NGRP.

UTEST (INPUT/OUTPUT, OPTIONAL) real(stnd) On entry, a probability. UTEST is the sum of the areas (equal) in both tails of the normal-distribution. UTEST must verify: 0. < P < 1.

On exit, the two-tail quantile of the normal-distribution, that is a value X such that the probability of the absolute value of U being greater than X is UTEST under the null hypothesis that the groups have been formed by uniform random sampling without duplication in the set of all the observations.

Further Details

The subroutine computes all the statistics with only one pass through the data.

If fewer than one valid observation were present for all variables, the statistics are set to Nan code.

If fewer than one valid observation were present only for some variables and/or groups of observations, the pertinent statistics are set to missing (XMISS value).

The optional parameters need only to be specified when LAST=true.

For more details, on the statistics and tests computed by this subroutine, see:

(1) **Terray, P., Delecluse, P., Labattu, S., Terray, L., 2003:** Sea Surface Temperature associations with the Late Indian Summer Monsoon. Climate Dynamics, vol. 21, 593-618.

6.22.54 `function valmed ( x )`

Purpose

Find the median of the vector X(:).

Arguments

X (INPUT) real(stnd), dimension(:) On entry, the vector of observations.

Further Details

This subroutine uses a modified quicksort algorithm.

6.22.55 `function valmed ( x )`

Purpose

Find the medians of the column vectors of the matrix X(:,:).

Arguments

X (INPUT) real(stnd), dimension(:,:) On entry, the matrix of observations.

Further Details

This subroutine uses a modified quicksort algorithm.

6.23 Module_Statpack

Copyright 2022 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

INTERFACE MODULE EXPORTING ALL PUBLIC CONSTANTS, VARIABLES, SUBROUTINES AND FUNCTIONS FROM OTHER MODULES AVAILABLE IN STATPACK.

LATEST REVISION : 18/07/2022

6.24 Module_String_Procedures

Copyright 2024 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING ROUTINES AND PARAMETERS FOR STRING PROCESSING AND PRINTING.

LATEST REVISION : 12/03/2024

6.24.1 `function ascii_is_upper ( c )`

Purpose

This function tests if the character C is an upper case letter.

Arguments

C (INPUT) character The character to test

Further Details

It uses the ASCII collating sequence.

6.24.2 `function is_upper ( c )`

Purpose

This function tests if the character C is an upper case letter.

Arguments

C (INPUT) character The character to test

Further Details

It uses the underlying machine collating sequence.

6.24.3 function ascii_is_lower (c)

Purpose

This function tests if the character C is a lower case letter.

Arguments

C (INPUT) character The character to test

Further Details

It uses the ASCII collating sequence.

6.24.4 function is_lower (c)

Purpose

This function tests if the character C is a lower case letter.

Arguments

C (INPUT) character The character to test

Further Details

It uses the underlying machine collating sequence.

6.24.5 function ascii_is_alpha (c)

Purpose

This function tests if the character C is a letter.

Arguments

C (INPUT) character The character to test

Further Details

It uses the ASCII collating sequence.

6.24.6 function is_alpha (c)

Purpose

This function tests if the character C is a letter.

Arguments

C (INPUT) character The character to test

Further Details

It uses the underlying machine collating sequence.

6.24.7 function ascii_is_same (c1, c2)

Purpose

ascii_is_same tests if C1 is the same character as C2 regardless of case.

Arguments

C1, C2 (INPUT) character The characters to test

Further Details

It uses the ASCII collating sequence.

6.24.8 function is_same (c1, c2)

Purpose

is_same tests if C1 is the same character as C2 regardless of case.

Arguments

C1, C2 (INPUT) character The characters to test

Further Details

It uses the underlying machine collating sequence.

6.24.9 function ascii_is_digit (c)

Purpose

This function tests if the character C is a digit.

Arguments

C (INPUT) character The character to test

Further Details

It uses the ASCII collating sequence.

6.24.10 function is_digit (c)

Purpose

This function tests if the character C is a digit.

Arguments

C (INPUT) character The character to test

Further Details

It uses the underlying machine collating sequence.

6.24.11 function is_space (c)

Purpose

This function tests if the character C is a space or a tabulation.

Arguments

C (INPUT) character The character to test

6.24.12 function is_num (string)

Purpose

This function tests if the character argument STRING contains a numerical value.

Arguments

STRING (INPUT) character(len=*) The string to analyze.

Further Details

Specifically, IS_NUM returns:

- KCHR = 0_i1b if STRING is a non-numerical string
- KINT = 1_i1b if STRING is an integer
- KFIX = 2_i1b if STRING is a fixed real
- KEXP = 3_i1b if STRING is a real with exponent

Definitions of KCHR, KINT, KFIX and KEXP may be obtained from the host module Strings.

This function is adapted from:

(1) **Olagnon, M., 1996:** Traitement de donnees numeriques avec Fortran 90. Masson, 264 pages, Chapter 5.3.2, ISBN 2-225-85259-6.

6.24.13 function string_count (string, letter)

Purpose

STRING_COUNT returns the number of occurences of the letter LETTER in the string STRING.

Comparison is case-sensitive and trailing blanks are ignored.

Arguments

STRING (INPUT) character(len=*) The string input.

LETTER (INPUT) character(len=1) The letter to compare against.

Further Details

The result is an integer of kind i4b.

6.24.14 function ascii_string_eq (string1, string2)

Purpose

ASCII_STRING_EQ tests if 2 strings are equal, ignoring case and trailing blanks.

Arguments

STRING1, STRING2 (INPUT) character(len=*) The strings to test.

Further Details

It uses the ASCII collating sequence.

6.24.15 function string_eq (string1, string2)

Purpose

STRING_EQ tests if 2 strings are equal, ignoring case and trailing blanks.

Arguments

STRING1, STRING2 (INPUT) character(len=*) The strings to test.

Further Details

It uses the underlying machine collating sequence.

6.24.16 function ascii_string_index (string, list)

Purpose

ASCII_STRING_INDEX returns index of a string in a list of strings, or 0 if no match. Comparison is case-insensitive and trailing blanks are ignored.

Arguments

STRING (INPUT) character(len=*) The string input.

LIST (INPUT) character(len=*), dimension(:) The list to compare against.

Further Details

It uses the ASCII collating sequence.

The result is an integer of kind i4b.

6.24.17 function string_index (string, list)

Purpose

STRING_INDEX returns index of a string in a list of strings, or 0 if no match. Comparison is case-insensitive and trailing blanks are ignored.

Arguments

STRING (INPUT) character(len=*) The string input.

LIST (INPUT) character(len=*), dimension(:) The list to compare against.

Further Details

It uses the underlying machine collating sequence.

The result is an integer of kind i4b.

6.24.18 function ascii_string_comp (string1, string2)

Purpose

ASCII_STRING_COMP compares 2 strings, ignoring case and trailing blanks. It returns:

- 0 if strings are equal,
- -1 if STRING1 < STRING2,
- +1 if STRING1 > STRING2.

Arguments

STRING1, STRING2 (INPUT) character(len=*) The strings to test.

Further Details

It uses the ASCII collating sequence.

The result is an integer of kind i1b.

6.24.19 function string_comp (string1, string2)

Purpose

STRING_COMP compares 2 strings, ignoring case and trailing blanks. It returns:

- 0 if strings are equal,
- -1 if STRING1 < STRING2,
- +1 if STRING1 > STRING2.

Arguments

STRING1, STRING2 (INPUT) character(len=*) The strings to test.

Further Details

It uses the underlying machine collating sequence.

The result is an integer of kind i1b.

6.24.20 subroutine ebc2asc (ebc_str, asc_str, nchr)

Purpose

EBC2ASC translates the EBCDIC string EBC_STR into a ASCII string ASC_STR.

Arguments

EBC_STR (INPUT) character(len=*) The EBCDIC string to translate.

ASC_STR (OUTPUT) character(len=*) The translated string (ASCII).

NCHR (INPUT) integer(i4b) Number of characters to convert.

Further Details

EBC2ASC assumes that the storage unit for default characters is one byte and that the storage unit for integers is a given number of bytes.

Non ASCII characters are translated as the "null" character (achar(0)).

This subroutine is adapted from:

(1) **Olagnon, M., 1996:** Traitement de donnees numeriques avec Fortran 90. Masson, 264 pages, Chapter 5.2, ISBN 2-225-85259-6.

6.24.21 subroutine asc2ebc (asc_str, ebc_str, nchr)

Purpose

ASC2EBC translates the ASCII string ASC_STR into a EBCDIC string EBC_STR .

Arguments

ASC_STR (INPUT) character(len=*) The ASCII string to translate.

EBC_STR (OUTPUT) character(len=*) The translated string (EBCDIC).

NCHR (INPUT) integer(i4b) Number of characters to convert.

Further Details

ASC2EBC assumes that the storage unit for default characters is one byte and that the storage unit for integers is a given number of bytes.

For machines with extended ASCII characters set or machines returning values greater than 127 for intrinsic function IACHAR(), ASC2EBC assumes that characters "c" with IACHAR(c)>127 are the same as ACHAR(IACHAR(c)-128).

This subroutine is adapted from:

(1) **Olagnon, M., 1996:** Traitement de donnees numeriques avec Fortran 90. Masson, 264 pages, Chapter 5.2, ISBN 2-225-85259-6.

6.24.22 function ascii_to_upper (c)

Purpose

This function converts the character C to upper case.

Arguments

C (INPUT) character

Further Details

All non-alphabetic characters are left unchanged. It uses the ASCII collating sequence.

6.24.23 function to_upper (c)

Purpose

This function converts the character C to upper case.

Arguments

C (INPUT) character

Further Details

All non-alphabetic characters are left unchanged. It uses the underlying machine collating sequence.

6.24.24 function ascii_to_lower (c)

Purpose

This function converts the character C to lower case.

Arguments

C (INPUT) character

Further Details

All non-alphabetic characters are left unchanged. It uses the ASCII collating sequence.

6.24.25 function to_lower (c)

Purpose

This function converts the character C to lower case.

Arguments

C (INPUT) character

Further Details

All non-alphabetic characters are left unchanged. It uses the underlying machine collating sequence.

6.24.26 subroutine ascii_case_change (string, type)

Purpose

This converts each lower case alphabetic letter in STRING to upper case, or vice versa.

Arguments

STRING (INPUT/OUTPUT) character(len=*) The string to convert

TYPE (INPUT/OUTPUT) integer(i1b) Define the conversion. Specifically, if:

- TYPE = 1_i1b = TOUPPER, conversion is lower to upper
- TYPE = 2_i1b = TOLOWER, conversion is upper to lower
- TYPE = 3_i1b = CAPITALIZE, use upper for first letter; lower for rest

Definitions of TOUPPER, TOLOWER and CAPITALIZE may be obtained from the host module Strings.

Further Details

All non-alphabetic characters are left unchanged. It uses the ASCII collating sequence.

6.24.27 subroutine case_change (string, type)

Purpose

This converts each lower case alphabetic letter in STRING to upper case, or vice versa.

Arguments

string (INPUT/OUTPUT) character(len=*) The string to convert

TYPE (INPUT/OUTPUT) integer(i1b) Define the conversion. Specifically, if:

- TYPE = 1_i1b = TOUPPER, conversion is lower to upper
- TYPE = 2_i1b = TOLOWER, conversion is upper to lower
- TYPE = 3_i1b = CAPITALIZE, use upper for first letter; lower for rest

Definitions of TOUPPER, TOLOWER and CAPITALIZE may be obtained from the host module Strings.

Further Details

All non-alphabetic characters are left unchanged. It uses the underlying machine collating sequence.

6.24.28 subroutine mid_shift (string, from, to, number)

Purpose

This routine performs a shift of characters within STRING. The number of characters shifted is NUMBER and they are shifted so that the character in position FROM is moved to position TO. Characters in the TO position are overwritten. Blanks replace characters in the FROM position. Shifting may be left or right, and the FROM and TO positions may overlap. Care is taken not to alter or use any characters beyond the defined limits of STRING.

Arguments

STRING (INPUT/OUTPUT) character(len=*) The string to modify.

FROM, TO, NUMBER (INPUT/OUTPUT) integer(i4b) Parameters defining the shift.

6.24.29 subroutine center (string)

Purpose

This routine shifts the nonblank characters of STRING so that there is a balance of blanks on left and right.

Arguments

STRING (INPUT/OUTPUT) character(len=*) The string to center.

6.24.30 subroutine find_field (string, istart, iend, delims, isearch)

Purpose

This routine returns the starting and ending positions of a delimitted field in STRING taking into account the set of delimitters stored in the string DELIMS.

Arguments

STRING (INPUT) character(len=*) The string to analyze.

ISTART (OUTPUT) integer(i4b) The starting position of the field.

IEND (OUTPUT) integer(i4b) The ending position of the field.

DELIMS (INPUT, OPTIONAL) character(len=*) The string containing the characters to be accepted as delimitters.

ISEARCH (INPUT, OPTIONAL) integer(i4b) The starting position for searching for the field.

Further Details

If the optional argument DELIMS is absent, the default delimitter set is a blank. If the optional argument ISEARCH is absent, the starting position for searching for the field is the first character of STRING.

On return, if:

- ISTART=0, STRING is empty, i.e contains only delimitters or is the null string ;
- ISTART/=0, the delimitted field is STRING(ISTART:IEND).

6.24.31 function nbrchf (jval)

Purpose

This function determines the number of characters (digits and sign) needed to represent the integer JVAL.

Arguments

JVAL (INPUT) integer(i4b) The integer to edit.

Further Details

The result is an integer of kind i4b.

6.24.32 function nbrchf (rval)

Purpose

This function determines the number of characters (digits and sign) needed to represent the integer part of RVAL.

Arguments

RVAL (INPUT) real(stnd) The real to edit.

Further Details

The result is an integer of kind i4b.

6.24.33 function obt_fmt (jval)

Purpose

This function determines the "(Iw)" format needed to edit the integer JVAL without excess blanks. The format is returned as a fixed length blank-padded string of 12 characters.

Arguments

JVAL (INPUT) integer(i4b) The integer to edit.

Further Details

If there is an error, FMT_INT returns a "blank string".

The result is character string of length 12.

6.24.34 function obt_fmt (rval, d)

Purpose

This function determines the "(Fw.d)" format needed to edit the real RVAL without excess blanks. The format is returned as a fixed length blank-padded string of 22 characters.

Arguments

RVAL (INPUT) real(stnd) The real to edit.

D (INPUT, OPTIONAL) integer(i4b) The desired number of decimal digits after the decimal point.

Further Details

If the optional parameter D is absent, the format returns by this function will edit the real RVAL with five decimals digits after the decimal point.

If there is an error, FMT_REAL returns a "blank" string.

The result is character string of length 22.

6.24.35 subroutine val_to_string (jval, string, nchar)

Purpose

This function converts an integer to a string. The value is returned left adjusted in the string.

Arguments

JVAL (INPUT) integer(i4b) The integer to convert.

STRING (OUTPUT) character(len=*) The string.

NCHAR (OUTPUT) integer(i4b) Number of characters used or needed to edit JVAL. The value is in STRING(1:NCHAR).

Further Details

If there is an error, INT_TO_STRING returns a string filled with '*' and the right length of string needed to edit JVAL in NCHAR.

6.24.36 subroutine val_to_string (rval, string, nchar, fmt, d)

Purpose

This function converts the real RVAL to a string with a given "Fw.d" or "Gw.d" format. The value is returned left adjusted in the string.

Arguments

RVAL (INPUT) real(stnd) The real to convert.

STRING (OUTPUT) character(len=*) The string.

NCHAR (OUTPUT) integer(i4b) Number of characters used or needed to edit RVAL. The value is in STRING(1:NCHAR).

FMT (INPUT,OPTIONAL) character "G" or "g" to use an G edit descriptor, an F edit descriptor is used for other values of FMT.

D (INPUT, OPTIONAL) integer(i4b) Number of digits to appear after the decimal point in the output field for an F edit descriptor or number of significant digits to print for an G edit descriptor.

Further Details

If the optional parameter FMT is absent, an F edit descriptor is used by default.

If the optional parameter D is absent, five decimals digits will appear after the decimal point in the string for an F edit descriptor or five significant digits will be printed if an G edit descriptor is used.

If there is an error, REAL_TO_STRING returns a string filled with "*" and the right length of string needed to edit RVAL in NCHAR.

6.24.37 subroutine string_to_val (string, kcode, fmt)

Purpose

This routine tests if the character argument STRING contains a numerical value and returns a format to read the string.

Arguments

STRING (INPUT) character(len=*) The string to analyze.

KCODE (OUTPUT) integer(i1b) KCODE is equal to:

- KCHR = 0_i1b if STRING is a non-numerical string
- KINT = 1_i1b if STRING is an integer
- KFIX = 2_i1b if STRING is a fixed real
- KEXP = 3_i1b if STRING is a real with exponent

Definitions of KCHR, KINT, KFIX and KEXP may be obtained from the host module Strings.

FMT (OUTPUT) character(len=14) The format to read the string.

Further Details

If there is an error, STRING_TO_VAL returns a format filled with blanks.

6.25 Module_The_Kinds

Copyright 2018 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING SYMBOLIC NAMES FOR KINDS OF LOGICAL, INTEGER, REAL AND COMPLEX TYPES AVAILABLE ON THE COMPUTER.

THE SYMBOLIC NAMES AVAILABLE AND EXPORTED BY THIS MODULE ARE DEFINED AS FOLLOW:

SYMBOLIC NAME FOR DEFAULT KIND OF LOGICAL:

integer, parameter :: logic = kind(.true.)

SYMBOLIC NAMES FOR KIND TYPES OF LOGICAL:

- integer, parameter :: logic0 = 0
- integer, parameter :: logic1 = 1
- integer, parameter :: logic2 = 2
- integer, parameter :: logic4 = 4

SYMBOLIC NAMES FOR KIND TYPES OF 1-, 2-, 4- and 8-BYTES INTEGERS:

- integer, parameter :: i1b = selected_int_kind(2)
- integer, parameter :: i2b = selected_int_kind(4)
- integer, parameter :: i4b = selected_int_kind(9)
- integer, parameter :: i8b = selected_int_kind(10)

SYMBOLIC NAMES FOR KIND TYPES OF SINGLE-, DOUBLE- and QUADRUPLE-PRECISION REAL AND COMPLEX NUMBERS:

- integer, parameter :: sp = kind(1.0)
- integer, parameter :: dp = kind(1.0d0)
- integer, parameter :: qp = selected_real_kind(precision(1.0d0) + 1)

THE qp KIND TYPE MAY NOT BE AVAILABLE ON YOUR COMPUTER.

PRECISION SPECIFICATIONS FOR REAL AND COMPLEX COMPUTATIONS:

- integer, parameter :: low = selected_real_kind(6, 35)
- integer, parameter :: normal = selected_real_kind(12, 50)
- integer, parameter :: extended = selected_real_kind(20, 80)

THESE PRECISION SPECIFICATIONS REQUEST, RESPECTIVELY, 6, 12, 20 DECIMAL DIGITS OF PRECISION AND AN EXPONENT RANGE OF AT LEAST 10 ^ +- 35, 10 ^ +- 50 AND 10 ^ +- 80. THE extended PRECISION MAY NOT BE AVAILABLE ON YOUR COMPUTER.

TO TEST THE AVAILABLE KIND TYPES AND PRECISIONS ON YOUR COMPUTER, YOU CAN USE THE PROGRAM test_kind.F90 (e.g. TYPE THE COMMAND "make test_kind" IN THE MAIN STATPACK DIRECTORY).

THE CHOICE BETWEEN THESE DIFFERENT KIND TYPES FOR COMPILING A VERSION OF STATPACK IS DONE IN THE MODULE Select_Parameters (AVAILABLE IN THE SOURCE FILE Module_Select_Parameters.F90).

LATEST REVISION : 23/04/2018

6.26 Module_Time_Procedures

Copyright 2024 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING TIME AND DATE UTILITIES.

LATEST REVISION : 19/02/2024

6.26.1 `function leapyr ( iyr )`

Purpose

Check for a leap year. LEAPYR is returned as "true" if IYR is a leap year, and "false" otherwise.

Arguments

IYR (INPUT) integer(i4b) The year to test.

Further Details

This function uses the Gregorian calendar adopted the Oct. 15, 1582.

Leap years are years that are evenly divisible by 4, except years that are evenly divisible by 100 must be divisible by 400.

The result is a logical of kind lgl.

6.26.2 `function daynum ( iyr, imon, iday )`

Purpose

Compute a day number. One of the more useful applications for this routine is to compute the number of days between two dates.

Arguments

IYR, IMON , IDAY (INPUT) integer(i4b) Year (iyr), month (imon), and day (iday).

Further Details

This function uses the Gregorian calendar adopted the Oct. 15, 1582.

In other words, Oct. 15, 1582 will return a day number of unity and hence this algorithm will not work properly for dates early than 10-15-1582.

The result is an integer of kind i4b.

6.26.3 `function day_of_week ( iyr, imon, iday )`

Purpose

This function returns the day of the week (e.g., Mon, Tue,...) as an index (Mon=1 to Sun=7) for a given year, month, and day.

Arguments

IYR, IMON , IDAY (INPUT) integer(i4b) Year (IYR), month (IMON), and day (IDAY).

Further Details

This routine assumes a valid day, month and year are input.

The result is an integer of kind i4b.

The algorithm is adapted from:

(1) **Larson, K., 1995:** Computing the Day of the Week. Dr Dobb's Journal, p. 125-126, April.

6.26.4 subroutine daynum_to_ymd (jdaynum, iyr, imon, iday)

Purpose

Converts a Julian Day Number (JDAYNUM) to Gregorian year (IYR), month (IMON) and day (IDAY).

Arguments

JDAYNUM (INPUT) integer(i4b) The Julian day number to convert. See further details.

IYR, IMON , IDAY (OUTPUT) integer(i4b) The year (IYR), month (IMON), and day (IDAY) in the Gregorian calendar corresponding to the Julian day number JDAYNUM.

Further Details

This subroutine converts the integer JDAYNUM to three integers IYR, IMON and IDAY standing for year, month, day in the Gregorian calendar promulgated by Gregory XIII, starting with JDAYNUM=1 on Friday, 15 October 1582.

To keep Pope Gregory's calendar synchronized with the seasons for the next 16000 years or so, a small correction has been introduced; millennial years divisible by 4000 are not considered leap-years.

Note the Gregorian calendar was adopted in Oct. 15, 1582, and hence this algorithm will not work properly for dates early than 10-15-1582, e.g. if JDAYNUM < 1.

Note that England and its possessions remained 10 or 11 days behind this Gregorian calendar until 14 September 1752, e.g. if JDAYNUM < 62062.

Note that no consensus has been reached yet about dates beyond Monday, 28 February, 4000 (e.g. if JDAYNUM > 882928), and dates after 18000 A.D. are extremely speculative at best.

This subroutine is adapted from a MATLAB M-file written by W. Kahan available on the WEB.

6.26.5 function ymd_to_daynum (iyr, imon, iday)

Purpose

Converts Gregorian year (IYR), month (IMON) and day (IDAY) to Julian day Number. See further details.

Arguments

IYR, IMON , IDAY (INPUT) integer(i4b) The year (IYR), month (IMON), and day (IDAY) in the Gregorian calendar to convert.

Further Details

This function converts the three integers IYR, IMON and IDAY standing for year, month, day in the Gregorian calendar promulgated by Gregory XIII on Friday, 15 October 1582, in the corresponding Julian day number starting with YMD_TO_DAYNUM=1 on Friday, 15 October 1582.

Note the Gregorian calendar was adopted in Oct. 15, 1582, and hence this algorithm will not work properly for dates early than 10-15-1582. Dates are checked for validity by using DAYNUM_TO_YMD subroutine.

The number of days between two dates is the difference between their Julian day.

The result is an integer of kind i4b.

This subroutine is adapted from a MATLAB M-file written by W. Kahan available on the WEB.

6.26.6 function ymd_to_dayweek (iyr, imon, iday)

Purpose

Computes the day of the week from Gregorian year (IYR), month (IMON) and day (IDAY). See further details.

Arguments

IYR, IMON , IDAY (INPUT) integer(i4b) The year (IYR), month (IMON), and day (IDAY) in the Gregorian calendar to convert.

Further Details

This function returns the day of the week (e.g., Mon, Tue,...) as an integer index (Mon=1 to Sun=7) for the given year, month, and day in the Gregorian calendar promulgated by Gregory XIII on Friday, 15 October 1582.

Note that the Gregorian calendar was adopted in Oct. 15, 1582, and hence this algorithm will not work properly for dates early than 10-15-1582. Dates are checked for validity by using DAYNUM_TO_YMD subroutine.

The result is an integer of kind i4b.

This subroutine is adapted from a MATLAB M-file written by W. Kahan available on the WEB.

6.26.7 function daynum_to_dayweek (jdaynum)

Purpose

Computes the day of the week from Julian day number JDAYNUM. See further details.

Arguments

JDAYNUM (INPUT) integer(i4b) The Julian day number to convert.

Further Details

This function returns the day of the week (e.g., Mon, Tue,...) as an integer index (Mon=1 to Sun=7) for the given Julian day number JDAYNUM starting with JDAYNUM=1 on Friday, 15 October 1582.

The result is an integer of kind i4b.

6.26.8 function rtsw ()

Purpose

RTSW is a Real-Time Stop Watch.

This routine can be used to compute the time lapse between functions calls according to the system (wall) clock.

Arguments

None.

Further Details

RTSW is a unique numeric identifier derived from the current system time and date.

This function works across month and year boundaries, but is not thread-safe and cannot be called in parallel by different threads.

Since this routine uses the system clock, the elapsed time computed with this routine may not (probably won't be in a multi-tasking OS) an accurate reflection of the number of cpu cycles required to perform a calculation. Therefore care should be exercised when using this to profile a code.

The result is a real of kind extd.

The calling procedure for this function is as follow:

tim1 = rtsw()

[perform calculations]

tim2 = rtsw()

write(,) 'Elapsed Time (s): ',tim2-tim1

6.26.9 function elapsed_time (t1, t0)

Purpose

Computes elapsed time between two invocations of the intrinsic function DATE_AND_TIME. ELAPSED_TIME(T1, T0) returns the time in seconds that has elapsed between the vectors T0 and T1. Each vector must have at least seven elements in the format returned by DATE_AND_TIME for the optional argument VALUES; namely

T = (/ year, month, day, x, hour, minute, second /)

This routine can be used to compute the elapsed time between DATE_AND_TIME calls according to the system (wall) clock.

Arguments

T1, T0 (INPUT) integer, dimension(:) The two vectors which give the starting and ending dates and times as returned by DATE_AND_TIME.

Further Details

Since this routine uses the system clock, the elapsed time computed with this routine may not (probably won't be in a multi-tasking OS) an accurate reflection of the number of cpu cycles required to perform a calculation. Therefore care should be exercised when using this to profile a code.

This function works across month and year boundaries but does not check the validity of its arguments, which are expected to be obtained as in the following example that shows how to time some operation by using ELAPSED_TIME.

The result is an integer of kind i4b.

The calling procedure for this subroutine is as follow:

```
call date_and_time( values=t0(:) )
    [perform calculations]
call date_and_time( values=t1(:) )
write(,) 'Elapsed Time (s): ', elapsed_time( t1(:), t0(:) )
```

This subroutine is adapted from a MATLAB M-file written by W. Kahan available on the WEB.

6.26.10 function cpusecs ()

Purpose

This function obtains, from the intrinsic routine SYSTEM_CLOCK, the current value of the system CPU usage clock. This value is then converted to seconds and returned as an extended precision real value.

Arguments

None.

Further Details

This functions assumes that the number of CPU cycles (clock counts) between two calls is less than COUNT_MAX, the maximum possible value of clock counts as returned by the intrinsic routine SYSTEM_CLOCK.

The result is a real of kind extd.

The calling procedure for this function is as follow:

```
tim1 = cpusecs()
    [perform calculations]
tim2 = cpusecs()
write(,) 'CPU Time (s): ',tim2-tim1
```

6.26.11 subroutine time_to_hmsms (time, hmsms)

Purpose

Convert time to hours, minutes, seconds, milliseconds format.

Arguments

TIME (INPUT) real(extd) The time in seconds.

HMSMS (OUTPUT) integer(i4b), dimension(4) On exit, an integer array with:

- HMSMS(1) = the hours as an integer number
- HMSMS(2) = the minutes as an integer number from 0 to 59
- HMSMS(3) = the seconds as an integer number from 0 to 59
- HMSMS(4) = the milliseconds as an integer number from 0 to 999

6.26.12 function time_to_string (time)

Purpose

Convert TIME to a string format for printing as

'hours.minutes.seconds.milliseconds'

Arguments

TIME (INPUT) real(extd) The time in seconds.

Further Details

The result is a string of (at least) 13 characters.

6.26.13 subroutine get_date (iyr, imon, iday, date)

Purpose

Get a given date in a "nice" format.

Arguments

IYR, IMON , IDAY (INPUT) integer(i4b) Year (IYR), month (IMON), and day (IDAY).

DATE (OUTPUT) character(len=*) The date in the form dd-mmm-yyyy as in 18-Mar-1992.

Should be at least 11 chars long to hold the full string.

Further Details

If DATE is more than 11 characters in length, DATE is padded with blanks. If it is less than 11 characters in length, only the leftmost characters of the date will be returned.

If there is an error, GET_DATE returns a string filled with '*'.

6.26.14 subroutine get_date_time (date, time)

Purpose

Get system date and time in "nice" formats. This routine just reformats the output from the standard DATE_AND_TIME intrinsic.

Arguments

DATE (OUTPUT,OPTIONAL) character(len=*) The date in the form dd-mmm-yyyy as in 18-Mar-1992.

Should be at least 11 chars long to hold full string.

TIME (OUTPUT,OPTIONAL) character(len=*) The time in the form hh:mm:ss.

Should be at least 8 chars long to hold full string.

Further Details

If DATE is more than 11 characters in length, DATE is padded with blanks. If it is less than 11 characters in length, only the leftmost characters of the date will be returned.

If TIME is more than 8 characters in length, TIME is padded with blanks. If it is less than 8 characters in length, only the leftmost characters of the time will be returned.

6.26.15 subroutine system_date_time (chdate)

Purpose

Retrieve the current system time and date and transfer them to CHDATE in a "pretty" format, i.e.,

"DATE: DD-MMM-YYYY TIME: HH:MM:SS"

Arguments

CHDATE (OUTPUT) character(len=*) The string to hold the time and the date.

Should be at least 33 chars long to hold full string.

Further Details

If CHDATE is more than 33 characters in length, CHDATE is padded with blanks. If it is less than 33 in length, only the leftmost characters of the date will be returned.

6.26.16 subroutine my_date_time (chdate)

Purpose

This routine returns in CHDATE a 41-character date of the form given in model (below). It uses the time and date as obtained from the intrinsic routine DATE_AND_TIME and converts them to the form of the model given below:

'00:00 a.m., Wednesday, September 00, 1999'

Arguments

CHDATE (OUTPUT) character(len=*) The string to hold the time and the date. Should be at least 41 chars long to hold full string.

Further Details

Note that excess blanks in the date are eliminated. If CHDATE is more than 41 characters in length, CHDATE is padded with blanks. If it is less than 41 in length, only the leftmost characters of the date will be returned.

6.27 Module_Time_Series_Procedures

Copyright 2020 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING SUBROUTINES AND FUNCTIONS FOR TIME SERIES ANALYSIS

LATEST REVISION : 16/09/2020

6.27.1 `subroutine comp_smooth ( x, smooth_factor )`

Purpose

Smooth a time series.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:) On entry, input vector containing size(X) observations which must be smoothed with a smoothing factor of SMOOTH_FACTOR. On exit, the smoothed vector.

SMOOTH_FACTOR (INPUT) integer(i4b) On entry, the smoothing factor. The smoothing factor must be greater than 0 and less than size(X).

Further Details

The input vector is smoothed with a moving average of, approximately, (2 * SMOOTH_FACTOR) + 1 terms.

For further details, see:

(1) **Olagnon, M., 1996:** Traitement de donnees numeriques avec Fortran 90. Masson, 264 pages, Chapter 11.1.2, ISBN 2-225-85259-6.

6.27.2 `subroutine comp_smooth ( x, smooth_factor, dimvar )`

Purpose

Smooth the rows or the columns of a matrix.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, input matrix containing size(X,3-DIMVAR) observations on size(X,DIMVAR) variables which must be smoothed with a smoothing factor of SMOOTH_FACTOR. By default, DIMVAR is equal to 1. See description of optional DIMVAR argument for details.

SMOOTH_FACTOR (INPUT) integer(i4b) On entry, the smoothing factor. The smoothing factor must be greater than 0 and less than size(X,3-DIMVAR).

DIMVAR (INPUT, OPTIONAL) integer(i4b) On entry, if DIMVAR is present, DIMVAR is used as follows:

- DIMVAR = 1, the input matrix X contains size(X,2) observations on size(X,1) variables and the rows of X will be smoothed.
- DIMVAR = 2, the input submatrix X contains size(X,1) observations on size(X,2) variables and the columns of X will be smoothed.

The default is DIMVAR = 1.

Further Details

The input matris is smoothed along the specified dimension with a moving average of, approximately, (2 * SMOOTH_FACTOR) + 1 terms.

For further details, see:

(1) **Olagnon, M., 1996:** Traitement de donnees numeriques avec Fortran 90. Masson, 264 pages, Chapter 11.1.2, ISBN 2-225-85259-6.

6.27.3 `subroutine comp_smooth ( x, smooth_factor )`

Purpose

Smooth a tridimensional array along the third dimension.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:,:,:) On entry, input tridimensional array containing size(X,3) observations on size(X,1) by size(X,2) variables which must be smoothed with a smoothing factor of SMOOTH_FACTOR.

SMOOTH_FACTOR (INPUT) integer(i4b) On entry, the smoothing factor. The smoothing factor must be greater than 0 and less than size(X,3).

Further Details

The input tridimensional array is smoothed along the third dimension with a moving average of, approximately, (2 * SMOOTH_FACTOR) + 1 terms.

For further details, see:

(1) **Olagnon, M., 1996:** Traitement de donnees numeriques avec Fortran 90. Masson, 264 pages, Chapter 11.1.2, ISBN 2-225-85259-6.

6.27.4 subroutine comp_trend (y, nt, itdeg, robust, trend, ntjump, maxiter, rw, no, ok)

Purpose

COMP_TREND extracts a smoothed component from a time series using a LOESS method. It returns the smoothed component (e.g. the trend) and, optionally, the robustness weights.

Arguments

Y (INPUT) real(stnd), dimension(:) On entry, the time series to be decomposed.

NT (INPUT) integer(i4b) On entry, the length of the trend smoother. The value of NT should be an odd integer greater than or equal to 3. As NT increases the values of the trend component become smoother.

ITDEG (INPUT) integer(i4b) On entry, the degree of locally-fitted polynomial in trend smoothing. The value must be 0, 1 or 2.

ROBUST (INPUT) logical(lgl) On entry, TRUE if robustness iterations are to be used, FALSE otherwise.

Robustness iterations are carried out until convergence of the trend component, with MAXITER iterations maximum. Convergence occurs if the maximum changes in the trend fit is less than 1% of the component's range after the previous iteration.

TREND (OUTPUT) real(stnd), dimension(:) On output, the smoothed (e.g. trend) component.

TREND must verify: size(TREND) = size(Y).

NTJUMP (INPUT, OPTIONAL) integer(i4b) On entry, the skipping value for trend smoothing. By default, NTJUMP is set to NT/10.

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, the maximum number of robustness iterations.

The default is 15. This argument is not used if ROBUST=FALSE.

RW (OUTPUT, OPTIONAL) real(stnd), dimension(:) On output, final robustness weights. All RW elements are 1 if ROBUST=FALSE .

RW must verify: size(RW) = size(Y).

NO (OUTPUT, OPTIONAL) integer(i4b) On output, if:

- ROBUST=TRUE : the number of robustness iterations. The iterations end if a convergence criterion is met or if the number is MAXITER.
- ROBUST=FALSE : NO is set to 0.

OK (OUTPUT, OPTIONAL) logical(lgl) On output, if:

- ROBUST=TRUE : OK is set to TRUE if the convergence criterion is met and to FALSE otherwise.
- ROBUST=FALSE : OK is set to TRUE.

Further Details

This subroutine is adapted from subroutine STL developped by Cleveland and coworkers at AT&T Bell Laboratories.

This subroutine decomposes a time series into trend and residual components, assuming that the time series has no seasonal cycle or other harmonic components. The algorithm uses LOESS interpolation to smooth the time series and find the trend.

The LOESS smoother for estimating the trend is specified with three parameters: a width (e.g. NT), a degree (e.g. ITDEG) and a jump (e.g. NTJUMP). The width specifies the number of data points that the local interpolation uses to smooth each point, the degree specifies the degree of the local polynomial that is fit to the data, and the jump specifies how many points are skipped between Loess interpolations, with linear interpolation being done between these points.

If the optional ROBUST argument is set to true, the process is iterative and includes robustness iterations that take advandages of the weighted-least-squares underpinnings of LOESS to remove the effects of outliers.

Note that, finally, that this subroutine expects equally spaced data with no missing values.

For further details, see:

(1) **Cleveland, R.B., Cleveland, W.S., McRae, J.E., and Terpenning, I.:** STL: A Seasonal-Trend Decomposition Procedure Based on Loess. Statistics Research Report, AT&T Bell Laboratories.

(2) **Cleveland, R.B., Cleveland, W.S., McRae, J.E., and Terpenning, I., 1990:** STL: A Seasonal-Trend Decomposition Procedure Based on Loess. J. Official Stat., 6, 3-73.

(3) **Crotinger, J., 2017:** Java implementation of Seasonal-Trend-Loess time-series decomposition algorithm. https://github.com/ServiceNow/stl-decomp-4j

6.27.5 subroutine comp_trend (y, nt, itdeg, robust, trend, ntjump, maxiter, rw, no, ok)

Purpose

COMP_TREND extracts smoothed components from the (time series) columns of a matrix using a LOESS method. It returns the smoothed components (e.g. the trends) and, optionally, the robustness weights.

Arguments

Y (INPUT) real(stnd), dimension(:,:) On entry, the matrix to be decomposed.

NT (INPUT) integer(i4b) On entry, the length of the trend smoother. The value of NT should be an odd integer greater than or equal to 3. As NT increases the values of the trend component become smoother.

ITDEG (INPUT) integer(i4b) On entry, the degree of locally-fitted polynomial in trend smoothing. The value must be 0, 1 or 2.

ROBUST (INPUT) logical(lgl) On entry, TRUE if robustness iterations are to be used, FALSE otherwise.

Robustness iterations are carried out until convergence of the trend component, with MAXITER iterations maximum. Convergence occurs if the maximum changes in the trend fit is less than 1% of the component's range after the previous iteration.

TREND (OUTPUT) real(stnd), dimension(:,:) On output, the smoothed (e.g. trend) components.

TREND must verify: size(TREND,1) = size(Y,1) and size(TREND,2) = size(Y,2).

NTJUMP (INPUT, OPTIONAL) integer(i4b) On entry, the skipping value for trend smoothing.

By default, NTJUMP is set to NT/10.

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, the maximum number of robustness iterations.

The default is 15. This argument is not used if ROBUST=FALSE.

RW (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On output, final robustness weights. All RW elements are 1 if ROBUST=FALSE .

RW must verify: size(RW,1) = size(Y,1) and size(RW,2) = size(Y,2).

NO (OUTPUT, OPTIONAL) integer(i4b), dimension(:) On output, if

- ROBUST=TRUE : NO(i) is the number of robustness iterations for each time series Y(:,i). The iterations end if a convergence criterion is met or if the number is MAXITER for each time series Y(:,i).
- ROBUST=FALSE : NO(:) is set to 0.

NO must verify: size(NO) = size(Y,2).

OK (OUTPUT, OPTIONAL) logical(lgl), dimension(:) On output, if

- ROBUST=TRUE : OK(i) is set to TRUE if the convergence criterion is met for time series Y(:,i) and to FALSE otherwise.
- ROBUST=FALSE : OK(:) is set to TRUE.

OK must verify: size(OK) = size(Y,2).

Further Details

This subroutine is adapted from subroutine STL developped by Cleveland and coworkers at AT&T Bell Laboratories.

This subroutine decomposes a multi-channel time series into trend and residual components, assuming that the multi-channel time series has no seasonal cycle or other harmonic components. The algorithm uses LOESS interpolation to smooth the multi-channel time series and find the trends.

The LOESS smoother for estimating the trends is specified with three parameters: a width (e.g. NT), a degree (e.g. ITDEG) and a jump (e.g. NTJUMP). The width specifies the number of data points that the local interpolation uses to smooth each point, the degree specifies the degree of the local polynomial that is fit to the data, and the jump specifies how many points are skipped between Loess interpolations, with linear interpolation being done between these points.

If the optional ROBUST argument is set to true, the process is iterative and includes robustness iterations that take advandages of the weighted-least-squares underpinnings of LOESS to remove the effects of outliers.

Note that, finally, that this subroutine expects equally spaced data with no missing values.

For further details, see description of COMP_STL and :

(1) **Cleveland, R.B., Cleveland, W.S., McRae, J.E., and Terpenning, I.:** STL: A Seasonal-Trend Decomposition Procedure Based on Loess. Statistics Research Report, AT&T Bell Laboratories.

(2) **Cleveland, R.B., Cleveland, W.S., McRae, J.E., and Terpenning, I., 1990:** STL: A Seasonal-Trend Decomposition Procedure Based on Loess. J. Official Stat., 6, 3-73.

(3) **Crotinger, J., 2017:** Java implementation of Seasonal-Trend-Loess time-series decomposition algorithm. https://github.com/ServiceNow/stl-decomp-4j

6.27.6 subroutine comp_stlez (y, np, ns, isdeg, itdeg, robust, season, trend, ni, nt, nl, ildeg, nsjump, ntjump, nljump, maxiter, rw, no, ok)

Purpose

COMP_STLEZ decomposes a time series into seasonal and trend components. It returns the components and, optionally, the robustness weights.

COMP_STLEZ offers an easy to use version of COMP_STL subroutine, also included in STATPACK, by defaulting most parameters values associated with the three LOESS smoothers used in COMP_STL.

Arguments

Y (INPUT) real(stnd), dimension(:) On entry, the time series to be decomposed.

NP (INPUT) integer(i4b) On entry, the period of the seasonal component. For example, if the time series is monthly with a yearly cycle, then NP=12 should be used. NP must be greater than 1.

NS (INPUT) integer(i4b) On entry, the length of the seasonal smoother. The value of NS should be an odd integer greater than or equal to 3; NS>6 is recommended. As NS increases the values of the seasonal component at a given point in the seasonal cycle (e.g., January values of a monthly series with a yearly cycle) become smoother.

ISDEG (INPUT) integer(i4b) On entry, the degree of locally-fitted polynomial in seasonal smoothing. The value must be 0 or 1.

ITDEG (INPUT) integer(i4b) On entry, the degree of locally-fitted polynomial in trend smoothing. The value must be 0, 1 or 2.

ROBUST (INPUT) logical(lgl) On entry, TRUE if robustness iterations are to be used, FALSE otherwise.

Robustness iterations are carried out until convergence of both seasonal and trend components, with MAXITER iterations maximum. Convergence occurs if the maximum changes in individual seasonal and trend fits are less than 1% of the component's range after the previous iteration.

SEASON (OUTPUT) real(stnd), dimension(:) On output, the seasonal component.

SEASON must verify: size(SEASON) = size(Y).

TREND (OUTPUT) real(stnd), dimension(:) On output, the trend component.

TREND must verify: size(TREND) = size(Y).

NI (INPUT, OPTIONAL) integer(i4b) On entry, the number of loops for updating the seasonal and trend components. The value of NI should be a positive integer.

By default, NI=2 if ROBUST=FALSE and NI=1 if ROBUST=TRUE.

NT (INPUT, OPTIONAL) integer(i4b) On entry, the length of the trend smoother. The value of NT should be an odd integer greater than or equal to 3. A value of NT between 1.5 * NP and 2 * NP is recommended. As NT increases the values of the trend component become smoother.

By default, NT is set to the smallest odd integer greater than or equal to (1.5 * NP) / (1-(1.5/NS)).

NL (INPUT, OPTIONAL) integer(i4b) On entry, the length of the low-pass filter. The value of NL should be an odd integer greater than or equal to 3. The smallest odd integer greater than or equal to NP is recommended.

By default, NL is set to the smallest odd integer greater than or equal to NP.

ILDEG (INPUT, OPTIONAL) integer(i4b) On entry, the degree of locally-fitted polynomial in low-pass smoothing.

By default, ILDEG is set to ITDEG.

NSJUMP (INPUT, OPTIONAL) integer(i4b) On entry, the skipping value for seasonal smoothing. The seasonal smoother skips ahead NSJUMP points and then linearly interpolates in between. The value of NSJUMP should be a positive integer; if NSJUMP=1, a seasonal smooth is calculated at all size(Y) points. To make the procedure run faster, a reasonable choice for NSJUMP is 10% or 20% of NS.

By default, NSJUMP is set to NS/10.

NTJUMP (INPUT, OPTIONAL) integer(i4b) On entry, the skipping value for trend smoothing.

By default, NTJUMP is set to NT/10.

NLJUMP (INPUT, OPTIONAL) integer(i4b) On entry, the skipping value for the low-pass filter.

By default, NLJUMP is set to NL/10.

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, the maximum number of robustness iterations.

The default is 15. This argument is not used if ROBUST=FALSE.

RW (OUTPUT, OPTIONAL) real(stnd), dimension(:) On output, final robustness weights. All RW elements are 1 if ROBUST=FALSE .

RW must verify: size(RW) = size(Y).

NO (OUTPUT, OPTIONAL) integer(i4b) On output, if

- ROBUST=TRUE : the number of robustness iterations. The iterations end if a convergence criterion is met or if the number is MAXITER.
- ROBUST=FALSE : NO is set to 0.

OK (OUTPUT, OPTIONAL) logical(lgl) On output, if

- ROBUST=TRUE : OK is set to TRUE if the convergence criterion is met and to FALSE otherwise.
- ROBUST=FALSE : OK is set to TRUE.

Further Details

This subroutine is a FORTRAN90 implementation of subroutine STLEZ developped by Cleveland and coworkers at AT&T Bell Laboratories.

At a minimum, COMP_STLEZ requires specifying the periodicity of the data (e.g. NP, 12 for monthly), the width of the LOESS smoother used to smooth the cyclic seasonal sub-series (e.g. NS) and the degree of the locally-fitted polynomial in seasonal (e.g. ISDEG) and trend (e.g. ITDEG) smoothing.

COMP_STLEZ sets, by default, others parameters of the STL procedure to the values recommended in Cleveland et al. (1990). It also includes tests of convergence if robust iterations are carried out. Otherwise, COMP_STLEZ is similar to COMP_STL.

For further details, see description of COMP_STL and:

(1) **Cleveland, R.B., Cleveland, W.S., McRae, J.E., and Terpenning, I.:** STL: A Seasonal-Trend Decomposition Procedure Based on Loess. Statistics Research Report, AT&T Bell Laboratories.

(2) **Cleveland, R.B., Cleveland, W.S., McRae, J.E., and Terpenning, I., 1990:** STL: A Seasonal-Trend Decomposition Procedure Based on Loess. J. Official Stat., 6, 3-73.

(3) **Crotinger, J., 2017:** Java implementation of Seasonal-Trend-Loess time-series decomposition algorithm. https://github.com/ServiceNow/stl-decomp-4j

6.27.7 subroutine comp_stlez (y, np, ns, isdeg, itdeg, robust, season, trend, ni, nt, nl, ildeg, nsjump, ntjump, nljump, maxiter, rw, no, ok)

Purpose

COMP_STLEZ decomposes the (time series) columns of a matrix into seasonal and trend components. It returns the components and, optionally, the robustness weights.

COMP_STLEZ offers an easy to use version of COMP_STL subroutine, also included in STATPACK, by defaulting most parameters values associated with the three LOESS smoothers used in COMP_STL.

Arguments

Y (INPUT) real(stnd), dimension(:,:) On entry, the matrix to be decomposed.

NP (INPUT) integer(i4b) On entry, the period of the seasonal component. For example, if the time series is monthly with a yearly cycle, then NP=12 should be used. NP must be greater than 1.

NS (INPUT) integer(i4b) On entry, the length of the seasonal smoother. The value of NS should be an odd integer greater than or equal to 3; NS>6 is recommended. As NS increases the values of the seasonal component at a given point in the seasonal cycle (e.g., January values of a monthly series with a yearly cycle) become smoother.

ISDEG (INPUT) integer(i4b) On entry, the degree of locally-fitted polynomial in seasonal smoothing. The value must be 0 or 1.

ITDEG (INPUT) integer(i4b) On entry, the degree of locally-fitted polynomial in trend smoothing. The value must be 0, 1 or 2.

ROBUST (INPUT) logical(lgl) On entry, TRUE if robustness iterations are to be used, FALSE otherwise. Robustness iterations are carried out until convergence of both seasonal and trend components, with MAXITER iterations maximum. Convergence occurs if the maximum changes in individual seasonal and trend fits are less than 1% of the component's range after the previous iteration.

SEASON (OUTPUT) real(stnd), dimension(:,:) On output, the seasonal components.

SEASON must verify: size(SEASON,1) = size(Y,1) and size(SEASON,2) = size(Y,2).

TREND (OUTPUT) real(stnd), dimension(:,:) On output, the trend components.

TREND must verify: size(TREND,1) = size(Y,1) and size(TREND,2) = size(Y,2).

NI (INPUT, OPTIONAL) integer(i4b) On entry, the number of loops for updating the seasonal and trend components. The value of NI should be a positive integer.

By default, NI=2 if ROBUST=FALSE and NI=1 if ROBUST=TRUE.

NT (INPUT, OPTIONAL) integer(i4b) On entry, the length of the trend smoother. The value of NT should be an odd integer greater than or equal to 3.

A value of NT between 1.5 * NP and 2 * NP is recommended. As NT increases the values of the trend component become smoother.

By default, NT is set to the smallest odd integer greater than or equal to (1.5 * NP) / (1-(1.5/NS)).

NL (INPUT, OPTIONAL) integer(i4b) On entry, the length of the low-pass filter. The value of NL should be an odd integer greater than or equal to 3.

The smallest odd integer greater than or equal to NP is recommended.

By default, NL is set to the smallest odd integer greater than or equal to NP.

ILDEG (INPUT, OPTIONAL) integer(i4b) On entry, the degree of locally-fitted polynomial in low-pass smoothing.

By default, ILDEG is set to ITDEG.

NSJUMP (INPUT, OPTIONAL) integer(i4b) On entry, the skipping value for seasonal smoothing. The seasonal smoother skips ahead NSJUMP points and then linearly interpolates in between. The value of NSJUMP should be a positive integer; if NSJUMP=1, a seasonal smooth is calculated at all size(Y) points. To make the procedure run faster, a reasonable choice for NSJUMP is 10% or 20% of NS.

By default, NSJUMP is set to NS/10.

NTJUMP (INPUT, OPTIONAL) integer(i4b) On entry, the skipping value for trend smoothing.

By default, NTJUMP is set to NT/10.

NLJUMP (INPUT, OPTIONAL) integer(i4b) On entry, the skipping value for the low-pass filter.

By default, NLJUMP is set to NL/10.

MAXITER (INPUT, OPTIONAL) integer(i4b) On entry, the maximum number of robustness iterations.

The default is 15. This argument is not used if ROBUST=FALSE.

RW (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On output, final robustness weights. All RW elements are 1 if ROBUST=FALSE .

RW must verify: size(RW,1) = size(Y,1) and size(RW,2) = size(Y,2).

NO (OUTPUT, OPTIONAL) integer(i4b), dimension(:) On output, if

- ROBUST=TRUE : NO(i) is the number of robustness iterations for each time series Y(:,i). The iterations end if a convergence criterion is met or if the number is MAXITER for each time series Y(:,i).
- ROBUST=FALSE : NO(:) is set to 0.

NO must verify: size(NO) = size(Y,2).

OK (OUTPUT, OPTIONAL) logical(lgl), dimension(:) On output, if

- ROBUST=TRUE : OK(i) is set to TRUE if the convergence criterion is met for time series Y(:,i) and to FALSE otherwise.
- ROBUST=FALSE : OK(:) is set to TRUE.

OK must verify: size(OK) = size(Y,2).

Further Details

This subroutine is a FORTRAN90 implementation of subroutine STLEZ developped by Cleveland and coworkers at AT&T Bell Laboratories.

At a minimum, COMP_STLEZ requires specifying the periodicity of the data (e.g. NP, 12 for monthly), the width of the LOESS smoother used to smooth the cyclic seasonal sub-series (e.g. NS) and the degree of the locally-fitted polynomial in seasonal (e.g. ISDEG) and trend (e.g. ITDEG) smoothing.

COMP_STLEZ sets, by default, others parameters of the STL procedure to the values recommended in Cleveland et al. (1990). It also includes tests of convergence if robust iterations are carried out. Otherwise, COMP_STLEZ is similar to COMP_STL.

For further details, see:

(1) **Cleveland, R.B., Cleveland, W.S., McRae, J.E., and Terpenning, I.:** STL: A Seasonal-Trend Decomposition Procedure Based on Loess. Statistics Research Report, AT&T Bell Laboratories.

(2) **Cleveland, R.B., Cleveland, W.S., McRae, J.E., and Terpenning, I., 1990:** STL: A Seasonal-Trend Decomposition Procedure Based on Loess. J. Official Stat., 6, 3-73.

(3) **Crotinger, J., 2017:** Java implementation of Seasonal-Trend-Loess time-series decomposition algorithm. https://github.com/ServiceNow/stl-decomp-4j

6.27.8 `subroutine comp_stl ( y, np, ni, no, isdeg, itdeg, ildeg, nsjump, ntjump, nljump, ns, nt, nl, rw, season, trend )`

Purpose

COMP_STL decomposes a time series into seasonal and trend components using LOESS smoothers. It returns the components and robustness weights.

Arguments

Y (INPUT) real(stnd), dimension(:) On entry, the time series to be decomposed.

NP (INPUT) integer(i4b) On entry, the period of the seasonal component. For example, if the time series is monthly with a yearly cycle, then NP=12 should be used. NP must be greater than 1.

NI (INPUT) integer(i4b) On entry, the number of loops for updating the seasonal and trend components. The value of NI should be a strictly positive integer.

NO (INPUT) integer(i4b) On entry, the number of robustness iterations. The value of NO should be a positive integer.

ISDEG (INPUT) integer(i4b) On entry, the degree of locally-fitted polynomial in seasonal smoothing. The value must be 0 or 1.

ITDEG (INPUT) integer(i4b) On entry, the degree of locally-fitted polynomial in trend smoothing. The value must be 0, 1 or 2.

ILDEG (INPUT) integer(i4b) On entry, the degree of locally-fitted polynomial in low-pass smoothing. The value must be 0, 1 or 2.

NSJUMP (INPUT/OUTPUT) integer(i4b) On entry, the skipping value for seasonal smoothing. The seasonal smoother skips ahead NSJUMP points and then linearly interpolates in between. The value of NSJUMP should be a positive integer; if NSJUMP=1, a seasonal smooth is calculated at all size(Y) points. To make the procedure run faster, a reasonable choice for NSJUMP is 10% or 20% of NS.

NTJUMP (INPUT/OUTPUT) integer(i4b) On entry, the skipping value for trend smoothing.

NLJUMP (INPUT/OUTPUT) integer(i4b) On entry, the skipping value for the low-pass filter.

NS (INPUT/OUTPUT) integer(i4b) On entry, the length of the seasonal smoother. The value of NS should be an odd integer greater than or equal to 3; NS>6 is recommended. As NS increases the values of the seasonal component at a given point in the seasonal cycle (e.g., January values of a monthly series with a yearly cycle) become smoother.

NT (INPUT/OUTPUT) integer(i4b) On entry, the length of the trend smoother. The value of NT should be an odd integer greater than or equal to 3. A value of NT between 1.5 * NP and 2 * NP is recommended. As NT increases the values of the trend component become smoother.

NL (INPUT/OUTPUT) integer(i4b) On entry, the length of the low-pass filter. The value of NL should be an odd integer greater than or equal to 3. The smallest odd integer greater than or equal to NP is recommended.

RW (OUTPUT) real(stnd), dimension(:) On output, final robustness weights. All RW elements are 1 if NO=0 .

RW must verify: size(RW) = size(Y).

SEASON (OUTPUT) real(stnd), dimension(:) On output, the seasonal component.

SEASON must verify: size(SEASON) = size(Y).

TREND (OUTPUT) real(stnd), dimension(:) On output, the trend component.

TREND must verify: size(TREND) = size(Y).

Further Details

This subroutine is a FORTRAN90 implementation of subroutine STL developped by Cleveland and coworkers at AT&T Bell Laboratories.

This subroutine decomposes a time series into seasonal, trend and residual components. The algorithm uses LOESS interpolation and smoothers to smooth the time series and estimate the seasonal (or harmonic) component and the trend. This process is iterative with many steps and may include robustness iterations that take advantage of the weighted-least-squares underpinnings of LOESS to remove the effects of outliers.

There are three LOESS smoothers in COMP_STL and each require three parameters: a width, a degree, and a jump. The width specifies the number of data points that the local interpolation uses to smooth each point, the degree specifies the degree of the local polynomial that is fit to the data, and the jump specifies how many points are skipped between LOESS interpolations, with linear interpolation being done between these points.

The LOESS smoother for estimating the trend is specified with the following parameters: a width (e.g. NT), a degree (e.g. ITDEG) and a jump (e.g. NTJUMP).

The LOESS smoother for estimating the seasonal component is specified with the following parameters: a width (e.g. NS), a degree (e.g. ISDEG) and a jump (e.g. NSJUMP).

The LOESS smoother for low-pass filtering is specified with the following parameters: a width (e.g. NL), a degree (e.g. ILDEG) and a jump (e.g. NLJUMP).

If the NO argument is set to an integer value greater than 0, the process includes also robustness iterations that take advandages of the weighted-least-squares underpinnings of LOESS to remove the effects of outliers.

Note that, finally, that this subroutine expects equally spaced data with no missing values.

For further details, see:

(1) **Cleveland, R.B., Cleveland, W.S., McRae, J.E., and Terpenning, I.:** STL: A Seasonal-Trend Decomposition Procedure Based on Loess. Statistics Research Report, AT&T Bell Laboratories.

(2) **Cleveland, R.B., Cleveland, W.S., McRae, J.E., and Terpenning, I., 1990:** STL: A Seasonal-Trend Decomposition Procedure Based on Loess. J. Official Stat., 6, 3-73.

(3) **Crotinger, J., 2017:** Java implementation of Seasonal-Trend-Loess time-series decomposition algorithm. https://github.com/ServiceNow/stl-decomp-4j

6.27.9 subroutine comp_stl (y, np, ni, no, isdeg, itdeg, ildeg, nsjump, ntjump, nljump, ns, nt, nl, rw, season, trend)

Purpose

COMP_STL decomposes the (time series) columns of a matrix into seasonal and trend components using LOESS smoothers. It returns the components and robustness weights.

Arguments

Y (INPUT) real(stnd), dimension(:,:) On entry, the time series to be decomposed.

NP (INPUT) integer(i4b) On entry, the period of the seasonal component. For example, if the time series is monthly with a yearly cycle, then NP=12 should be used. NP must be greater than 1.

NI (INPUT) integer(i4b) On entry, the number of loops for updating the seasonal and trend components. The value of NI should be a strictly positive integer.

NO (INPUT) integer(i4b) On entry, the number of robustness iterations. The value of NO should be a positive integer.

ISDEG (INPUT) integer(i4b) On entry, the degree of locally-fitted polynomial in seasonal smoothing. The value must be 0 or 1.

ITDEG (INPUT) integer(i4b) On entry, the degree of locally-fitted polynomial in trend smoothing. The value must be 0, 1 or 2.

ILDEG (INPUT) integer(i4b) On entry, the degree of locally-fitted polynomial in low-pass smoothing. The value must be 0, 1 or 2.

NSJUMP (INPUT/OUTPUT) integer(i4b) On entry, the skipping value for seasonal smoothing. The seasonal smoother skips ahead NSJUMP points and then linearly interpolates in between. The value of NSJUMP should be a positive integer; if NSJUMP=1, a seasonal smooth is calculated at all size(Y) points. To make the procedure run faster, a reasonable choice for NSJUMP is 10% or 20% of NS.

NTJUMP (INPUT/OUTPUT) integer(i4b) On entry, the skipping value for trend smoothing.

NLJUMP (INPUT/OUTPUT) integer(i4b) On entry, the skipping value for the low-pass filter.

NS (INPUT/OUTPUT) integer(i4b) On entry, the length of the seasonal smoother. The value of NS should be an odd integer greater than or equal to 3; NS>6 is recommended. As NS increases the values of the seasonal component at a given point in the seasonal cycle (e.g., January values of a monthly series with a yearly cycle) become smoother.

NT (INPUT/OUTPUT) integer(i4b) On entry, the length of the trend smoother. The value of NT should be an odd integer greater than or equal to 3. A value of NT between 1.5 * NP and 2 * NP is recommended. As NT increases the values of the trend component become smoother.

NL (INPUT/OUTPUT) integer(i4b) On entry, the length of the low-pass filter. The value of NL should be an odd integer greater than or equal to 3. The smallest odd integer greater than or equal to NP is recommended.

RW (OUTPUT) real(stnd), dimension(:,:) On output, final robustness weights. All RW elemets are 1 if NO=0 .

RW must verify: size(RW,1) = size(Y,1) and size(RW,2) = size(Y,2).

SEASON (OUTPUT) real(stnd), dimension(:,:) On output, the seasonal components.

SEASON must verify: size(SEASON,1) = size(Y,1) and size(SEASON,2) = size(Y,2).

TREND (OUTPUT) real(stnd), dimension(:,:) On output, the trend components.

TREND must verify: size(TREND,1) = size(Y,1) and size(TREND,2) = size(Y,2).

Further Details

This subroutine is a FORTRAN90 implementation of subroutine STL developped by Cleveland and coworkers at AT&T Bell Laboratories.

This subroutine decomposes a multi-channel time series into seasonal, trend and residual components. The algorithm uses LOESS interpolation and smoothers to smooth the multi-channel time series and estimate the seasonal (or harmonic) components and the trends. This process is iterative with many steps and may include robustness iterations that take advantage of the weighted-least-squares underpinnings of LOESS to remove the effects of outliers.

There are three LOESS smoothers in COMP_STL and each require three parameters: a width, a degree, and a jump. The width specifies the number of data points that the local interpolation uses to smooth each point, the degree specifies the degree of the local polynomial that is fit to the data, and the jump specifies how many points are skipped between LOESS interpolations, with linear interpolation being done between these points.

The LOESS smoother for estimating the trend is specified with the following parameters: a width (e.g. NT), a degree (e.g. ITDEG) and a jump (e.g. NTJUMP).

The LOESS smoother for estimating the seasonal component is specified with the following parameters: a width (e.g. NS), a degree (e.g. ISDEG) and a jump (e.g. NSJUMP).

The LOESS smoother for low-pass filtering is specified with the following parameters: a width (e.g. NL), a degree (e.g. ILDEG) and a jump (e.g. NLJUMP).

If the NO argument is set to an integer value greater than 0, the process includes also robustness iterations that take advandages of the weighted-least-squares underpinnings of LOESS to remove the effects of outliers.

Note that, finally, that this subroutine expects equally spaced data with no missing values.

For further details, see:

(1) **Cleveland, R.B., Cleveland, W.S., McRae, J.E., and Terpenning, I.:** STL: A Seasonal-Trend Decomposition Procedure Based on Loess. Statistics Research Report, AT&T Bell Laboratories.

(2) **Cleveland, R.B., Cleveland, W.S., McRae, J.E., and Terpenning, I., 1990:** STL: A Seasonal-Trend Decomposition Procedure Based on Loess. J. Official Stat., 6, 3-73.

(3) **Crotinger, J., 2017:** Java implementation of Seasonal-Trend-Loess time-series decomposition algorithm. https://github.com/ServiceNow/stl-decomp-4j

6.27.10 `subroutine ma ( x, len, ave)`

Purpose

Smooth the vector X with a moving average of length LEN and output the result in the vector AVE.

Arguments

X (INPUT) real(stnd), dimension(:) On entry, the vector to smooth.

LEN (INPUT) integer(i4b) On entry, the length of the moving average. The argument LEN must be >=1 and < size(x).

AVE (OUTPUT) real(stnd), dimension(size(x)) On output, AVE(1:size(X)-LEN+1) contains the smoothed values and AVE(size(X)-LEN+2:size(X)) is unchanged.

Further Details

This subroutine is a low-level subroutine used by subroutines COMP_STLEZ and COMP_STL.

6.27.11 `subroutine detrend ( vec, trend, orig, slope )`

Purpose

Subroutine DETREND detrends a time series (e.g. the argument VEC).

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) The time series vector to be detrended.

TREND (INPUT) integer(i4b) If:

- TREND=1 The mean of the time series is removed
- TREND=2 The drift from the time series is removed by using the formula:

  drift = (VEC(size(VEC)) - VEC(1))/(size(VEC) - 1)

- TREND=3 The least-squares line from the time series is removed.

For other values of TREND nothing is done.

ORIG (OUTPUT, OPTIONAL) real(stnd) On exit, the constant term if TREND=1 or 3.

SLOPE (OUTPUT, OPTIONAL) real(stnd) On exit, the linear term if TREND=2 or 3.

Further Details

On exit, the original time series may be recovered with the formula

VEC(i) = VEC(i) + ORIG + SLOPE * real(i-1,stnd)

for i=1, size(vec), in all the cases.

6.27.12 `subroutine detrend ( mat, trend, orig, slope )`

Purpose

Subroutine DETREND detrends a multi-channel time series (e.g. the argument MAT). Each row of matrix MAT is a real time series

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) The multi-channel time series matrix to be detrended.

TREND (INPUT) integer(i4b) If:

- TREND=1 The means of the time series are removed
- TREND=2 The drifts from the time series are removed by using the formula:

  drift(:) = (MAT(:,size(MAT,2)) - MAT(:,1))/(size(MAT,2) - 1)

- TREND=3 The least-squares lines from the time series are removed.

For other values of TREND nothing is done.

ORIG (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the constant terms if TREND=1 or 3.

The size of ORIG must verify: size(ORIG) = size(MAT,1) .

SLOPE (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the linear terms if TREND=2 or 3.

The size of SLOPE must verify: size(SLOPE) = size(MAT,1) .

Further Details

On exit, the original time series may be recovered with the formula

MAT(j,i) = MAT(j,i) + ORIG(j) + SLOPE(j) * real(i-1,stnd)

for i=1, size(MAT,2) and j=1, size(MAT,1), in all the cases.

6.27.13 subroutine hwfilter (vec, pl, ph, initfft, trend, win)

Purpose

Subroutine HWFILTER filters a time series (e.g. the argument VEC) in the frequency band limited by periods PL and PH by windowed filtering (PL and PH are expressed in number of points, i.e. PL=6(18) and PH=32(96) selects periods between 1.5 yrs and 8 yrs for quarterly (monthly) data).

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) The time series vector to be filtered.

Size(VEC) must be greater or equal to 4.

PL (INPUT) integer(i4b) Minimum period of oscillation of desired component. Use PL=0 for high-pass filtering frequencies corresponding to periods shorter than PH, PL must be equal to 0 or greater or equal to 2. Moreover, PL must be less or equal to size(VEC).

PH (INPUT) integer(i4b) Maximum period of oscillation of desired component. USE PH=0 for low-pass filtering frequencies corresponding to periods longer than PL. PH must be equal to 0 or greater or equal to 2. Moreover, PH must be less or equal to size(VEC).

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if INITFFT is set to false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine HWFILTER in order to sets up constants and functions for use by subroutine FFT_ROW which is called inside subroutine HWFILTER (the call to INIT_FFT must have the following form:

call init_fft(size(VEC))

If INITFFT is set to true, the call to INIT_FFT is done inside subroutine HWFILTER and a call to END_FFT is also done before leaving subroutine HWFILTER.

The default is INITFFT=true.

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+/-1 The mean of the time series is removed before time filtering
- TREND=+/-2 The drift from the time series is removed before time filtering by using the formula: drift = (VEC(size(VEC)) - VEC(1))/(size(VEC) - 1)
- TREND=+/-3 The least-squares line from the time series is removed before time filtering.

IF TREND=-1,-2 or -3, the mean, drift or least-squares line is reintroduced post-filtering, respectively. For other values of TREND nothing is done before or after filtering.

WIN (INPUT, OPTIONAL) real(stnd) By default, Hamming window filtering is used (i.e. WIN=0.54). SET WIN=0.5 for Hanning window or WIN=1 for rectangular window.

WIN must be greater or equal to O.5 and less or equal to 1.

Further Details

Use PL=0 for high-pass filtering frequencies corresponding to periods shorter than PH, or PH=0 for low-pass filtering frequencies corresponding to periods longer than PL.

Setting PH<PL is also allowed and performs band rejection of periods between PH and PL (i.e. in that case the meaning of the PL and PH arguments are reversed).

Examples:

For quarterly data: call hwfilter(vec, pl=6, ph=32) returns component with periods between 1.5 and 8 yrs.

For monthly data: call hwfilter(vec, pl=0, ph=24) returns component with all periods less than 2 yrs.

For more details and algorithm, see:

(1) **Iacobucci, A., and Noullez, A., 2005:** A Frequency Selective Filter for Short-Length Time Series. Computational Economics, 25,75-102.

6.27.14 subroutine hwfilter (mat, pl, ph, initfft, trend, win, max_alloc)

Purpose

Subroutine HWFILTER filters a multi-channel time series (e.g. the argument MAT) in the frequency band limited by periods PL and PH by windowed filtering (PL and PH are expressed in number of points, i.e. PL=6(18) and PH=32(96) selects periods between 1.5 yrs and 8 yrs for quarterly (monthly) data).

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) The multi-channel time series matrix to be filtered. Each column of MAT corresponds to one observation.

Size(MAT,2) must be greater or equal to 4.

PL (INPUT) integer(i4b) Minimum period of oscillation of desired component. Use PL=0 for high-pass filtering frequencies corresponding to periods shorter than PH, PL must be equal to 0 or greater or equal to 2. Moreover, PL must be less or equal to size(MAT,2).

PH (INPUT) integer(i4b) Maximum period of oscillation of desired component. USE PH=0 for low-pass filtering frequencies corresponding to periods longer than PL. PH must be equal to 0 or greater or equal to 2. Moreover, PH must be less or equal to size(MAT,2).

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if INITFFT is set to false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine HWFILTER in order to sets up constants and functions for use by subroutine FFT_ROW which is called inside subroutine HWFILTER (the call to INIT_FFT must have the following form:

call init_fft(shape(MAT), dim=2_i4b)

If INITFFT is set to true, the call to INIT_FFT is done inside subroutine HWFILTER and a call to END_FFT is also done before leaving subroutine HWFILTER.

The default is INITFFT=true.

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+/-1 The means of the time series are removed before time filtering

- TREND=+/-2 The drifts from the time series are removed before time filtering by using the formula: drift(:) = (MAT(:,size(MAT,2)) - MAT(:,1))/(size(MAT,2) - 1)
- TREND=+/-3 The least-squares lines from the time series are removed before time filtering.

IF TREND=-1,-2 or -3, the means, drifts or least-squares lines are reintroduced post-filtering, respectively. For other values of TREND nothing is done before or after filtering.

WIN (INPUT, OPTIONAL) real(stnd) By default, Hamming window filtering is used (i.e. WIN=0.54). SET WIN=0.5 for Hanning window or WIN=1 for rectangular window.

WIN must be greater or equal to O.5 and less or equal to 1.

MAX_ALLOC (INPUT, OPTIONAL) integer(i4b) MAX_ALLOC is a factor which allows to reduce the workspace used to compute the Fourier transform of the data if necessary at the expense of increasing the computing time. MAX_ALLOC must be greater or equal to 1 and less or equal to size(MAT,1).

The default is MAX_ALLOC= size(MAT,1).

Further Details

Use PL=0 for high-pass filtering frequencies corresponding to periods shorter than PH, or PH=0 for low-pass filtering frequencies corresponding to periods longer than PL.

Setting PH<PL is also allowed and performs band rejection of periods between PH and PL (i.e. in that case the meaning of the PL and PH arguments are reversed).

Examples:

For quarterly data: call hwfilter(mat, pl=6, ph=32) returns components with periods between 1.5 and 8 yrs.

For monthly data: call hwfilter(mat, pl=0, ph=24) returns components with all periods less than 2 yrs.

For more details and algorithm, see :

(1) **Iacobucci, A., and Noullez, A., 2005:** A Frequency Selective Filter for Short-Length Time Series. Computational Economics, 25,75-102.

6.27.15 `subroutine hwfilter2 ( vec, pl, ph, trend, win )`

Purpose

Subroutine HWFILTER2 filters a time series (e.g. the argument VEC) in the frequency band limited by periods PL and PH by windowed filtering (PL and PH are expressed in number of points, i.e. PL=6(18) and PH=32(96) selects periods between 1.5 yrs and 8 yrs for quarterly (monthly) data).

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) The time series vector to be filtered. Size(VEC) must be greater or equal to 4.

PL (INPUT) integer(i4b) Minimum period of oscillation of desired component. Use PL=0 for high-pass filtering frequencies corresponding to periods shorter than PH, PL must be equal to 0 or greater or equal to 2. Moreover, PL must be less or equal to size(VEC).

PH (INPUT) integer(i4b) Maximum period of oscillation of desired component. USE PH=0 for low-pass filtering frequencies corresponding to periods longer than PL. PH must be equal to 0 or greater or equal to 2. Moreover, PH must be less or equal to size(VEC).

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+/-1 The mean of the time series is removed before time filtering
- TREND=+/-2 The drift from the time series is removed before time filtering by using the formula: drift = (VEC(size(VEC)) - VEC(1))/(size(VEC) - 1)
- TREND=+/-3 The least-squares line from the time series is removed before time filtering.

IF TREND=-1,-2 or -3, the mean, drift or least-squares line is reintroduced post-filtering, respectively. For other values of TREND nothing is done before or after filtering.

WIN (INPUT, OPTIONAL) real(stnd) By default, Hamming window filtering is used (i.e. WIN=0.54). SET WIN=0.5 for Hanning window or WIN=1 for rectangular window.

WIN must be greater or equal to O.5 and less or equal to 1.

Further Details

Use PL=0 for high-pass filtering frequencies corresponding to periods shorter than PH, or PH=0 for low-pass filtering frequencies corresponding to periods longer than PL.

Setting PH<PL is also allowed and performs band rejection of periods between PH and PL (i.e. in that case the meaning of the PL and PH arguments are reversed).

The unique difference between HWFILTER2 and HWFILTER is the use of the Goertzel method for computing the Fourier transform of the data instead of a Fast Fourier Transform algorithm.

Examples:

For quarterly data: call hwfilter2(vec, pl=6, ph=32) returns component with periods between 1.5 and 8 yrs.

For monthly data: call hwfilter2(vec, pl=0, ph=24) returns component with all periods less than 2 yrs.

For more details and algorithm, see :

(1) **Iacobucci, A., and Noullez, A., 2005:** A Frequency Selective Filter for Short-Length Time Series. Computational Economics, 25,75-102.

(2) **Goertzel, G., 1958:** An Algorithm for the Evaluation of Finite Trigonometric Series. The American Mathematical Monthly, Vol. 65, No. 1, pp. 34-35

6.27.16 subroutine hwfilter2 (mat, pl, ph, trend, win)

Purpose

Subroutine HWFILTER2 filters a multi-channel time series (e.g. the argument MAT) in the frequency band limited by periods PL and PH by windowed filtering (PL and PH are expressed in number of points, i.e. PL=6(18) and PH=32(96) selects periods between 1.5 yrs and 8 yrs for quarterly (monthly) data).

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) The multi-channel time series matrix to be filtered. Each column of MAT corresponds to one observation.

Size(MAT,2) must be greater or equal to 4.

PL (INPUT) integer(i4b) Minimum period of oscillation of desired component. Use PL=0 for high-pass filtering frequencies corresponding to periods shorter than PH, PL must be equal to 0 or greater or equal to 2. Moreover, PL must be less or equal to size(MAT,2).

PH (INPUT) integer(i4b) Maximum period of oscillation of desired component. USE PH=0 for low-pass filtering frequencies corresponding to periods longer than PL. PH must be equal to 0 or greater or equal to 2. Moreover, PH must be less or equal to size(MAT,2).

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+/-1 The means of the time series are removed before time filtering
- TREND=+/-2 The drifts from the time series are removed before time filtering by using the formula: drift(:) = (MAT(:,size(MAT,2)) - MAT(:,1))/(size(MAT,2) - 1)
- TREND=+/-3 The least-squares lines from the time series are removed before time filtering.

IF TREND=-1,-2 or -3, the means, drifts or least-squares lines are reintroduced post-filtering, respectively. For other values of TREND nothing is done before or after filtering.

WIN (INPUT, OPTIONAL) real(stnd) By default, Hamming window filtering is used (i.e. WIN=0.54). SET WIN=0.5 for Hanning window or WIN=1 for rectangular window.

WIN must be greater or equal to O.5 and less or equal to 1.

Further Details

Use PL=0 for high-pass filtering frequencies corresponding to periods shorter than PH, or PH=0 for low-pass filtering frequencies corresponding to periods longer than PL.

Setting PH<PL is also allowed and performs band rejection of periods between PH and PL (i.e. in that case the meaning of the PL and PH arguments are reversed).

The unique difference between HWFILTER2 and HWFILTER is the use of the Goertzel method for computing the Fourier transform of the data instead of a Fast Fourier Transform algorithm.

Examples:

For quarterly data: call hwfilter2(mat, pl=6, ph=32) returns components with periods between 1.5 and 8 yrs.

For monthly data: call hwfilter2(mat, pl=0, ph=24) returns components with all periods less than 2 yrs.

For more details and algorithm, see :

(1) **Iacobucci, A., and Noullez, A., 2005:** A Frequency Selective Filter for Short-Length Time Series. Computational Economics, 25,75-102.

(2) **Goertzel, G., 1958:** An Algorithm for the Evaluation of Finite Trigonometric Series. The American Mathematical Monthly, Vol. 65, No. 1, pp. 34-35

6.27.17 function lp_coef (pl, k, fc, notest_fc)

Purpose

Function LP_COEF computes the K-term least squares approximation to an -ideal- low pass filter with cutoff period PL (e.g. cutoff frequency FC = 1/PL).

This filter has a transfer function with a transition band of width delta surrounding FC, where delta = 4 * pi/K when FC is expressed in radians.

Arguments

PL (INPUT) integer(i4b) Minimum period of oscillation of desired component. The corresponding cut-off frequency is FC=1/PL (i.e. filter has zero response in the interval [FC+1/K,Nyquist] and one response in the interval [0,FC-1/K].

PL must be greater than 2 and FC must verify the following inequalities:

- FC - 1/K >= 0
- FC + 1/K < 0.5

K (INPUT) integer(i4b) The number of filter terms to be computed. K must be greater or equal to 3 and odd.

FC (INPUT, OPTIONAL) real(stnd) The user chosen cutoff frequency in cycles per sample interval. If the optional argument FC is used, the PL argument is not used to determine the cutoff frequency.

FC must verify the following inequalities:

- FC - 1/K >= 0
- FC + 1/K < 0.5

NOTEST_FC (INPUT, OPTIONAL) logical(lgl) On input, if this optional logical argument is set to true, the two tests on the cutoff frequency (e.g. FC - 1/K >= 0 and FC + 1/K < 0.5) are bypassed. However, in that case, the cutoff frequency FC must still verify the inequalities 0 < FC < 0.5.

Further Details

Function LP_COEF computes symmetric linear low-pass filter coefficients using a least squares approximation to an ideal low-pass filter with convergence factors (i.e. Lanczos window) which reduce overshoot and ripple (Bloomfield, 1976).

This low-pass filter has a transfer function which changes from approximately one to zero in a transition band about the ideal cutoff frequency FC (FC=1/PL), that is from (FC - 1/K) to (FC + 1/K), as discussed in section 6.4 of Bloomfield (1976). The user must specify the cutoff period (or the cutoff frequency) and the number of filter coefficients, which must be odd.

The user must also choose the number of filter terms, K, so that (FC - 1/K) >= 0 and (FC + 1/K) < 0.5 if the optional logical argument NOTEST_FC is not used or is not set to true.

In addition, K must be chosen as a compromise between:

1) A sharp cutoff, that is, 1/K small; and
2) Minimizing the number of data points lost by the filtering operations (e.g. (K-1)/2 data points will be lost from each end of the series).

The subroutine returns the normalized low-pass filter coefficients.

This function is adapted from the STARPAC software developed by the National Institute of Standards and Technology (NIST). For more details and algorithm, see

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York, Chapter 6.

6.27.18 function lp_coef2 (pl, k, fc, win, notest_fc)

Purpose

Function LP_COEF2 computes the K-term least squares approximation to an -ideal- low pass filter with cutoff period PL (e.g. cutoff frequency FC = 1/PL) by windowed filtering (e.g. Hamming window is used).

This filter has a transfer function with a transition band of width delta surrounding FC, where delta = 4 * pi/K when FC is expressed in radians.

Arguments

PL (INPUT) integer(i4b) Minimum period of oscillation of desired component. The corresponding cut-off frequency is FC=1/PL (i.e. filter has zero response in the interval [FC+1/K,Nyquist] and one response in the interval [0,FC-1/K].

PL must be greater than two and FC must verify the following inequalities:

- FC - 1/K >= 0
- FC + 1/K < 0.5

K (INPUT) integer(i4b) The number of filter terms to be computed. K must be greater or equal to 3 and odd.

FC (INPUT, OPTIONAL) real(stnd) The user chosen cutoff frequency in cycles per sample interval. If the optional argument FC is used, the PL argument is not used to determine the cutoff frequency.

FC must verify the following inequalities:

- FC - 1/K >= 0
- FC + 1/K < 0.5

WIN (INPUT, OPTIONAL) real(stnd) By default, Hamming window filtering is used (i.e. WIN=0.54). Set WIN=0.5 for Hanning window or WIN=1 for rectangular window.

WIN must be greater or equal to O.5 and less or equal to 1, otherwise WIN is reset to 0.54.

NOTEST_FC (INPUT, OPTIONAL) logical(lgl) On input, if this optional logical argument is set to true, the two tests on the cutoff frequency (e.g. FC - 1/K >= 0 and FC + 1/K < 0.5) are bypassed. However, in that case, the cutoff frequency FC must still verify the inequalities 0 < FC < 0.5.

Further Details

Function LP_COEF2 computes symmetric linear low-pass filter coefficients using a least squares approximation to an ideal low-pass filter. The Hamming window is used to reduce overshoot and ripple in the transfert function of the ideal low-pass filter.

This low-pass filter has a transfer function which changes from approximately one to zero in a transition band about the ideal cutoff frequency FC (FC=1/PL), that is from (FC - 1/K) to (FC + 1/K), as discussed in section 6.4 of Bloomfield (1976). The user must specify the cutoff period (or the cutoff frequency) and the number of filter coefficients, which must be odd.

The user must also choose the number of filter terms, K, so that (FC - 1/K) >= 0 and (FC + 1/K) < 0.5 if the optional logical argument NOTEST_FC is not used or is not set to true.

The overshoot and the associated ripples in the ideal transfert function are reduced by the use of the Hamming window.

In addition, K must be chosen as a compromise between:

1) A sharp cutoff, that is, 1/K small; and

2) Minimizing the number of data points lost by the filtering operations (e.g. (K-1)/2 data points will be lost from each end of the series).

The subroutine returns the normalized low-pass filter coefficients.

For more details and algorithm, see

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York, Chapter 6.

6.27.19 `function hp_coef ( ph, k, fc, notest_fc )`

Purpose

Function HP_COEF computes the K-term least squares approximation to an -ideal- high pass filter with cutoff period PH (e.g. cutoff frequency FC = 1/PH).

This filter has a transfer function with a transition band of width delta surrounding FC, where delta = 4 * pi/K when FC is expressed in radians.

Arguments

PH (INPUT) integer(i4b) Maximum period of oscillation of desired component. The corresponding cutoff frequency is FC=1/PH (i.e. filter has one response in the interval [FC+1/K,Nyquist] and zero response in the interval [0,FC-1/K].

PH must be greater than two and FC must verify the following inequalities:

- FC - 1/K >= 0
- FC + 1/K < 0.5

K (INPUT) integer(i4b) The number of filter terms to be computed. K must be greater or equal to 3 and odd.

FC (INPUT, OPTIONAL) real(stnd) The user chosen cutoff frequency in cycles per sample interval. If the optional argument FC is used, the PH argument is not used to determine the cutoff frequency.

FC must verify the following inequalities:

- FC - 1/K >= 0
- FC + 1/K < 0.5

NOTEST_FC (INPUT, OPTIONAL) logical(lgl) On input, if this optional logical argument is set to true, the two tests on the cutoff frequency (e.g. FC - 1/K >= 0 and FC + 1/K < 0.5) are bypassed. However, in that case, the cutoff frequency FC must still verify the inequalities 0 < FC < 0.5.

Further Details

Function HP_COEF computes symmetric linear high-pass filter coefficients from the corresponding low-pass filter as given by function LP_COEF. This is equivalent to subtracting the low-pass filtered series from the original time series.

This high-pass filter has a transfer function which changes from approximately zero to one in a transition band about the ideal cutoff frequency FC (FC=1/PH), that is from (FC - 1/K) to (FC + 1/K), as discussed in section 6.4 of Bloomfield (1976). The user must specify the cutoff period (or the cutoff frequency) and the number of filter coefficients, which must be odd.

The user must also choose the number of filter terms, K, so that (FC - 1/K) >= 0 and (FC + 1/K) < 0.5 if the optional logical argument NOTEST_FC is not used or is not set to true.

In addition, K must be chosen as a compromise between:

1) A sharp cutoff, that is, 1/K small; and

2) Minimizing the number of data points lost by the filtering operations (e.g. (K-1)/2 data points will be lost from each end of the series).

The subroutine returns the high-pass filter coefficients.

This function is adapted from the STARPAC software developed by the National Institute of Standards and Technology (NIST).

For more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York, Chapter 6.

6.27.20 function hp_coef2 (ph, k, fc, win, notest_fc)

Purpose

Function HP_COEF2 computes the K-term least squares approximation to an -ideal- high pass filter with cutoff period PH (e.g. cutoff frequency FC = 1/PH) by windowed filtering (e.g. Hamming window is used).

This filter has a transfer function with a transition band of width delta surrounding FC, where delta = 4 * pi/K when FC is expressed in radians.

Arguments

PH (INPUT) integer(i4b) Maximum period of oscillation of desired component. The corresponding cutoff frequency is FC=1/PH (i.e. filter has one response in the interval [FC+1/K,Nyquist] and zero response in the interval [0,FC-1/K].

PH must be greater than two and FC must verify the following inequalities:

- FC - 1/K >= 0
- FC + 1/K < 0.5

K (INPUT) integer(i4b) The number of filter terms to be computed. K must be greater or equal to 3 and odd.

FC (INPUT, OPTIONAL) real(stnd) The user chosen cutoff frequency in cycles per sample interval. If the optional argument FC is used, the PH argument is not used to determine the cutoff frequency.

FC must verify the following inequalities:

- FC - 1/K >= 0
- FC + 1/K < 0.5

WIN (INPUT, OPTIONAL) real(stnd) By default, Hamming window filtering is used (i.e. WIN=0.54). Set WIN=0.5 for Hanning window or WIN=1 for rectangular window.

WIN must be greater or equal to O.5 and less or equal to 1, otherwise WIN is reset to 0.54.

NOTEST_FC (INPUT, OPTIONAL) logical(lgl) On input, if this optional logical argument is set to true, the two tests on the cutoff frequency (e.g. FC - 1/K >= 0 and FC + 1/K < 0.5) are bypassed. However, in that case, the cutoff frequency FC must still verify the inequalities 0 < FC < 0.5.

Further Details

Function HP_COEF2 computes symmetric linear high-pass filter coefficients from the corresponding low-pass filter as given by function LP_COEF2. This is equivalent to subtracting the low-pass filtered series from the original time series.

This high-pass filter has a transfer function which changes from approximately zero to one in a transition band about the ideal cutoff frequency FC (FC=1/PH), that is from (FC - 1/K) to (FC + 1/K), as discussed in section 6.4 of Bloomfield (1976). The user must specify the cutoff period (or the cutoff frequency) and the number of filter coefficients, which must be odd.

The user must also choose the number of filter terms, K, so that (FC - 1/K) >= 0 and (FC + 1/K) < 0.5 if the optional logical argument NOTEST_FC is not used or is not set to true.

The overshoot and the associated ripples in the ideal transfert function are reduced by the use of the Hamming window.

In addition, K must be chosen as a compromise between:

1) A sharp cutoff, that is, 1/K small; and
2) Minimizing the number of data points lost by the filtering operations (e.g. (K-1)/2 data points will be lost from each end of the series).

The subroutine returns the high-pass filter coefficients.

For more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York, Chapter 6.

6.27.21 function bd_coef (pl, ph, k, fch, fcl, notest_fc)

Purpose

Function BD_COEF computes the K-term least squares approximation to an -ideal- band pass filter with cutoff periods PL and PH (e.g. cutoff frequencies 1/PL and 1/PH).

PL and PH are expressed in number of points, i.e. PL=6(18) and PH=32(96) selects periods between 1.5 yrs and 8 yrs for quarterly(monthly) data).

Alternatively, the user can directly specify the two cutoff frequencies, FCL and FCH, corresponding to PL and PH.

Arguments

PL (INPUT) integer(i4b) Minimum period of oscillation of desired component. The corresponding cutoff frequency is 1/PL. PL must be greater than two and must verify the following inequalities:

- 1/PH + 1.3/(K+1) <= 1/PL - 1.3/(K+1)
- 1/PL + 1/K < 0.5

PH (INPUT) integer(i4b) Maximum period of oscillation of desired component. The corresponding cutoff frequency is 1/PH. PH must be greater than two and 1/PH must verify the following inequalities:

- 0 <= 1/PH - 1/K
- 1/PH + 1.3/(K+1) <= 1/PL - 1.3/(K+1)

K (INPUT) integer(i4b) The number of filter terms to be computed. K must be greater or equal to 3 and odd.

FCH (INPUT, OPTIONAL) real(stnd) The user chosen (low) cutoff frequency in cycles per sample interval. If the optional argument FCH is used, the PH argument is not used to determine the (low) cutoff frequency.

FCH must verify the following inequalities:

- 0 <= FCH - 1/K
- FCH + 1.3/(K+1) <= FCL - 1.3/(K+1)

FCL (INPUT, OPTIONAL) real(stnd) The user chosen (high) cutoff frequency in cycles per sample interval. If the optional argument FCL is used, the PL argument is not used to determine the cutoff (high) frequency.

FCL must verify the following inequalities:

- FCH + 1.3/(K+1) <= FCL - 1.3/(K+1)
- FCL + 1/K < 0.5

NOTEST_FC (INPUT, OPTIONAL) logical(lgl) On input, if this optional logical argument is set to true, the tests on the cutoff frequencies FCH and FCL (e.g. FCH - 1/K >= 0 and FCL + 1/K < 0.5) are bypassed. However, in that case FCH and FCL must still verify the inequalities FCH > 0, FCL < 0.5 and FCH + 1.3/(K+1) <= FCL - 1.3/(K+1) .

Further Details

Function BD_COEF computes symmetric linear band-pass filter coefficients using a least squares approximation to an ideal band-pass filter that has convergence factors which reduce overshoot and ripple (Bloomfield, 1976).

This band-pass filter is computed as the difference between two low-pass filters with cutoff frequencies 1/PH and 1/PL, respectively (or FCH and FCL).

This band-pass filter has a transfer function which changes from approximately zero to one and one to zero in the transition bands about the ideal cutoff frequencies 1/PH and 1/PL), that is from (1/PH - 1/K) to (1/PH + 1/K) and (1/PL - 1/K) to (1/PL + 1/K), respectively. The user must specify the two cutoff periods and the number of filter coefficients, which must be odd.

The user must also choose the number of filter terms, K, so that:

- 0<=(1/PH - 1/K)
- (1/PH + 1.3/(K+1)) <= (1/PL - 1.3/(K+1))
- (1/PL + 1/K) < 0.5

However, if the optional logical argument NOTEST_FC is used and is set to true, the two tests

- 0<=(1/PH - 1/K)
- (1/PL + 1/K) < 0.5

are bypassed.

In addition, K must be chosen as a compromise between:

1) A sharp cutoff, that is, 1/K small; and
2) Minimizing the number of data points lost by the filtering operations (e.g. (K-1)/2 data points will be lost from each end of the series).

The subroutine returns the difference between the two corresponding normalized low-pass filter coefficients as computed by function LP_COEF.

For more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York, Chapter 6.

(2) **Duchon, C., 1979:** Lanczos filtering in one and two dimensions. Journal of applied meteorology, vol. 18, 1016-1022.

6.27.22 function bd_coef2 (pl, ph, k, fch, fcl, win, notest_fc)

Purpose

Function BD_COEF2 computes the K-term least squares approximation to an -ideal- band pass filter with cutoff periods PL and PH (e.g. cutoff frequencies 1/PL and 1/PH) by windowed filtering (e.g. Hamming window is used).

PL and PH are expressed in number of points, i.e. PL=6(18) and PH=32(96) selects periods between 1.5 yrs and 8 yrs for quarterly(monthly) data).

Alternatively, the user can directly specify the two cutoff frequencies, FCL and FCH, corresponding to PL and PH.

Arguments

PL (INPUT) integer(i4b) Minimum period of oscillation of desired component. The corresponding cutoff frequency is 1/PL. PL must be greater than two and must verify the following inequalities:

- 1/PH < 1/PL
- 1/PL + 1/K < 0.5

PH (INPUT) integer(i4b) Maximum period of oscillation of desired component. The corresponding cutoff frequency is 1/PH. PH must be greater than two and 1/PH must verify the following inequalities:

- 0 <= 1/PH - 1/K

- 1/PH < 1/PL

K (INPUT) integer(i4b) The number of filter terms to be computed. K must be greater or equal to 3 and odd.

FCH (INPUT, OPTIONAL) real(stnd) The user chosen (low) cutoff frequency in cycles per sample interval. If the optional argument FCH is used, the PH argument is not used to determine the (low) cutoff frequency.

FCH must verify the following inequalities:

- 0 <= FCH - 1/K
- FCH < FCL

FCL (INPUT, OPTIONAL) real(stnd) The user chosen (high) cutoff frequency in cycles per sample interval. If the optional argument FCL is used, the PL argument is not used to determine the cutoff (high) frequency.

FCL must verify the following inequalities:

- FCH < FCL
- FCL + 1/K < 0.5

WIN (INPUT, OPTIONAL) real(stnd) By default, Hamming window filtering is used (i.e. WIN=0.54). Set WIN=0.5 for Hanning window or WIN=1 for rectangular window.

WIN must be greater or equal to O.5 and less or equal to 1, otherwise WIN is reset to 0.54.

NOTEST_FC (INPUT, OPTIONAL) logical(lgl) On input, if this optional logical argument is set to true, the tests on the cutoff frequencies FCH and FCL (e.g. FCH - 1/K >= 0 and FCL + 1/K < 0.5) are bypassed. However, in that case FCH and FCL must still verify the inequalities FCH > 0, FCL < 0.5 and FCH < FCL .

Further Details

Function BD_COEF2 computes symmetric linear band-pass filter coefficients using a least squares approximation to an ideal band-pass filter. The Hamming window is used to reduce overshoot and ripple in the transfert function of the ideal low-pass filter.

This band-pass filter is computed as the difference between two low-pass filters with cutoff frequencies 1/PH and 1/PL, respectively (or FCH and FCL).

This band-pass filter has a transfer function which changes from approximately zero to one and one to zero in the transition bands about the ideal cutoff frequencies 1/PH and 1/PL), that is from (1/PH - 1/K) to (1/PH + 1/K) and (1/PL - 1/K) to (1/PL + 1/K), respectively. The user must specify the two cutoff periods and the number of filter coefficients, which must be odd. The user must also choose the number of filter terms, K, so that:

- 0 <= (1/PH - 1/K)
- 1/PH < 1/PL
- (1/PL + 1/K) < 0.5

However, if the optional logical argument NOTEST_FC is used and is set to true, the two tests

- 0 <= (1/PH - 1/K)
- (1/PL + 1/K) < 0.5

are bypassed.

The overshoot and the associated ripples in the ideal transfert function are reduced by the use of the Hamming window.

In addition, K must be chosen as a compromise between:

1) A sharp cutoff, that is, 1/K small; and

2) Minimizing the number of data points lost by the filtering operations (e.g. (K-1)/2 data points will be lost from each end of the series).

The subroutine returns the difference between the two corresponding normalized low-pass filter coefficients as computed by function LP_COEF2.

For more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York, Chapter 6.

6.27.23 `function pk_coef ( freq, k, notest_freq )`

Purpose

Function PK_COEF computes the K-term least squares approximation to an -ideal- band pass filter with peak response near one at the single frequency FREQ (e.g. the peak response is at period=1/FREQ).

Arguments

FREQ (INPUT) real(stnd) The band pass filter will have unit response at the single frequency FREQ. FREQ is expressed in cycles per sample interval.

The frequency FREQ must also be greater or equal to (1.3/(K+1) + 1/K) and less than 0.5 - (1.3/(K+1) + 1/K).

K (INPUT) integer(i4b) The number of filter terms to be computed. K must be greater or equal to 3 and odd.

NOTEST_FREQ (INPUT, OPTIONAL) logical(lgl) On input, if this optional logical argument is set to true, the frequency FREQ must only be greater or equal to 1.3/(K+1) and less than 0.5 - 1.3/(K+1).

Further Details

Function PK_COEF computes symmetric linear band-pass filter coefficients using a least squares approximation to an ideal band-pass filter that has convergence factors which reduce overshoot and ripple (Bloomfield, 1976).

This band-pass filter is computed as the difference between two low-pass filters with cutoff frequencies FCL and FCH, respectively (Duchon, 1979).

This band-pass filter has a transfer function which changes from approximately zero to one and one to zero in the transition bands about the cutoff frequencies FCH and FCL, that is from (FCH - 1/K) to FREQ and FREQ to (FCL + 1/K), respectively. The user must specify the frequency FREQ with unit response and the number of filter coefficients, which must be odd. The user must also choose the number of filter terms, K, as a compromise between:

1) A sharp cutoff, that is, 1/K small; and

2) Minimizing the number of data points lost by the filtering operations (e.g. (K-1)/2 data points will be lost from each end of the series).

The subroutine computes the two cutoff frequencies FCL and FCH as described by Duchon (1979) and returns the difference between the two corresponding normalized low-pass filter coefficients as computed by function LP_COEF.

For more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York, Chapter 6.

(2) **Duchon, C., 1979:** Lanczos filtering in one and two dimensions. Journal of applied meteorology, vol. 18, 1016-1022.

6.27.24 function moddan_coef (k, smooth_param)

Purpose

This function computes the impulse response function (e.g. weights) corresponding to a number of applications of modified Daniell filters as done in subroutine MODDAN_FILTER.

Arguments

K (INPUT) integer(i4b) The number of filter weights to be computed. K must be equal to 2 * (2+sum(SMOOTH_PARAM(:)))- 1

SMOOTH_PARAM (INPUT) integer(i4b), dimension(:) The array of the half-lengths of the modified Daniell filters to be applied. All the values in SMOOTH_PARAM(:) must be greater than 0.

Size(SMOOTH_PARAM) must be greater or equal to 1.

Further Details

For definition, more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York, Chapter 6.

6.27.25 subroutine freq_func (nfreq, coef, freqr, four_freq, freq)

Purpose

Subroutine FREQ_FUNC computes the frequency response function (e.g. the transfer function) of the symmetric linear filter given by the argument COEF(:).

The frequency response function is computed at NFREQ frequencies regularly sampled between 0 and the Nyquist frequency if the optional logical argument FOUR_FREQ is not used or at the NFREQ Fourier frequencies 2 * pi * j/nfreq for j=0 to NFREQ-1 if this argument is used and set to true.

Arguments

NFREQ (INPUT) integer(i4b) The number of frequencies at which the frequency response function must be evaluated.

COEF (INPUT) real(stnd), dimension(:) The array of symmetric linear filter coefficients.

Size(COEF) must be greater or equal to 3 and odd.

FREQR (OUTPUT) real(stnd), dimension(NFREQ) On output, the frequency response function.

FOUR_FREQ (INPUT, OPTIONAL) logical(lgl) On input, if this argument is set to true the frequency response function is evaluated at the Fourier frequencies 2 * pi * j/nfreq for j=0 to NFREQ-1.

FREQ (OUTPUT, OPTIONAL) real(stnd), dimension(NFREQ) The NFREQ frequencies, in cycles per sample interval, at which the frequency response function are evaluated.

Further Details

For more details, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York, Chapter 6.

(2) **Oppenheim, A.V., and Schafer, R.W., 1999:** Discrete-Time Signal Processing. Second Edition. Prentice-Hall, New Jersey.

6.27.26 `subroutine symlin_filter ( vec, coef, trend, nfilt )`

Purpose

Subroutine SYMLIN_FILTER performs a symmetric filtering operation on an input time series (e.g. the argument VEC).

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On input, the vector containing the time series to be filtered. On output, the filtered time series is returned in VEC(:NFILT). Note that (size(COEF)-1)/2 data points will be lost from each end of the series, so that NFILT (NFILT= size(VEC) - size(COEF) + 1) time observations are returned and the remainig and ending part of VEC(:) is set to zero.

Size(VEC) must be greater or equal to 4.

COEF (INPUT) real(stnd), dimension(:) The array of symmetric linear filter coefficients.

Size(COEF) must be odd, greater or equal to 3 and less or equal to size(VEC).

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+/-1 The mean of the time series is removed before time filtering
- TREND=+/-2 The drift from the time series is removed before time filtering by using the formula: drift = (VEC(size(VEC)) - VEC(1))/(size(VEC) - 1)
- TREND=+/-3 The least-squares line from the time series is removed before time filtering.

IF TREND=-1,-2 or -3, the mean, drift or least-squares line is reintroduced post-filtering, respectively. For other values of TREND nothing is done before or after filtering.

NFILT (OUTPUT, OPTIONAL) integer(i4b) The number of time observations in the filtered time series. On output, NFILT= size(VEC) - size(COEF) + 1.

Further Details

The filtering is done in place and (size(COEF)-1)/2 observations will be lost from each end of the time series.

Note, also, that the filtered time series is shifted in time and is stored in VEC(1:NFILT) on output, with NFILT= size(VEC) - size(COEF) + 1.

The symmetric linear filter coefficients (e.g. the array COEF) can be computed with the help of functions LP_COEF, LP_COEF2, HP_COEF, HP_COEF2, BD_COEF and BD_COEF2.

6.27.27 subroutine symlin_filter (mat, coef, trend, nfilt)

Purpose

Subroutine SYMLIN_FILTER performs a symmetric filtering operation on an input multi-channel time series (e.g. the argument MAT).

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) The multi-channel time series matrix to be filtered. Each column of MAT corresponds to one observation. On output, the multi-channel filtered time series are returned in MAT(:,:NFILT). Note that (size(COEF)-1)/2 observations will be lost from each end of the multi-channel series, so that NFILT (NFILT= size(MAT,2) - size(COEF) + 1) time observations are returned and the remainig part of MAT(:,:) is set to zero.

Size(MAT,2) must be greater or equal to 4.

COEF (INPUT) real(stnd), dimension(:) The array of symmetric linear filter coefficients.

Size(COEF) must be odd, greater or equal to 3 and less or equal to size(MAT,2).

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+/-1 The means of the time series are removed before time filtering
- TREND=+/-2 The drifts from the time series are removed before time filtering by using the formula: drift(:) = (MAT(:,size(MAT,2)) - MAT(:,1))/(size(MAT,2) - 1)
- TREND=+/-3 The least-squares lines from the time series are removed before time filtering.

IF TREND=-1,-2 or -3, the means, drifts or least-squares lines are reintroduced post-filtering, respectively. For other values of TREND nothing is done before or after filtering.

NFILT (OUTPUT, OPTIONAL) integer(i4b) The number of time observations in the filtered multi-channel time series. On output, NFILT= size(MAT,2) - size(COEF) + 1.

Further Details

The filtering is done in place and (size(COEF)-1)/2 observations will be lost from each end of the multi-channel series.

Note, also, that the filtered multi-channel time series is shifted in time and is stored in MAT(:,1:NFILT) on output, with NFILT= size(MAT,2) - size(COEF) + 1.

The symmetric linear filter coefficients (e.g. the array COEF) can be computed with the help of functions LP_COEF, LP_COEF2, HP_COEF, HP_COEF2, BD_COEF and BD_COEF2.

6.27.28 subroutine symlin_filter2 (vec, coef, trend, usefft, initfft)

Purpose

Subroutine SYMLIN_FILTER2 performs a symmetric filtering operation on an input time series (e.g. the argument VEC).

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On input, the vector containing the time series to be filtered. On output, the filtered time series is returned.

Size(VEC) must be greater or equal to 4.

COEF (INPUT) real(stnd), dimension(:) The array of symmetric linear filter coefficients.

Size(COEF) must be odd, greater or equal to 3 and less or equal to size(VEC).

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+/-1 The mean of the time series is removed before time filtering
- TREND=+/-2 The drift from the time series is removed before time filtering by using the formula: drift = (VEC(size(VEC)) - VEC(1))/(size(VEC) - 1)
- TREND=+/-3 The least-squares line from the time series is removed before time filtering.

IF TREND=-1,-2 or -3, the mean, drift or least-squares line is reintroduced post-filtering, respectively. For other values of TREND nothing is done before or after filtering.

USEFFT (INPUT, OPTIONAL) logical(lgl) On input, if USEFFT is used and is set to true, the symmetric linear filter is applied to the argument VEC by using a Fast Fourier Transform and the convolution theorem.

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if INITFFT is set to false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine SYMLIN_FILTER2 in order to sets up constants and functions for use by subroutine FFT_ROW which is called inside subroutine SYMLIN_FILTER2 (the call to INIT_FFT must have the following form:

```
call init_fft( shape(VEC) )
```

If INITFFT is set to true, the call to INIT_FFT is done inside subroutine SYMLIN_FILTER2 and a call to END_FFT is also done before leaving subroutine SYMLIN_FILTER2. This optional argument has an effect only if argument USEFFT is used with the value true.

The default is INITFFT=true .

Further Details

No time observations will be lost, however the first and last (size(COEF)-1)/2 time observations are affected by end effects.

If USEFFT is used with the value true, the values at both ends of the output series are computed by assuming that the input series is part of a periodic sequence of period size(VEC). Otherwise, each end of the filtered time series is estimated by truncated the symmetric linear filter coefficients array.

The symmetric linear filter coefficients (e.g. the array COEF) can be computed with the help of functions LP_COEF, LP_COEF2, HP_COEF, HP_COEF2, BD_COEF and BD_COEF2.

6.27.29 subroutine symlin_filter2 (mat, coef, trend, usefft, initfft)

Purpose

Subroutine SYMLIN_FILTER2 performs a symmetric filtering operation on an input multi-channel time series (e.g. the argument MAT).

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) The multi-channel time series matrix to be filtered. Each column of MAT corresponds to one observation. On output, the multi-channel filtered time series are returned.

Size(MAT,2) must be greater or equal to 4.

COEF (INPUT) real(stnd), dimension(:) The array of symmetric linear filter coefficients.

Size(COEF) must be odd, greater or equal to 3 and less or equal to size(MAT,2).

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+/-1 The means of the time series are removed before time filtering
- TREND=+/-2 The drifts from the time series are removed before time filtering by using the formula: drift(:) = (MAT(:,size(MAT,2)) - MAT(:,1))/(size(MAT,2) - 1)
- TREND=+/-3 The least-squares lines from the time series are removed before time filtering.

IF TREND=-1,-2 or -3, the means, drifts or least-squares lines are reintroduced post-filtering, respectively. For other values of TREND nothing is done before or after filtering.

USEFFT (INPUT, OPTIONAL) logical(lgl) On input, if USEFFT is used and is set to true, the symmetric linear filter is applied to the argument VEC by using a Fast Fourier Transform and the convolution theorem.

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if INITFFT is set to false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine SYMLIN_FILTER2 in order to sets up constants and functions for use by subroutine FFT_ROW which is called inside subroutine SYMLIN_FILTER2 (the call to INIT_FFT must have the following form:

```
call init_fft( shape(MAT), dim=2_i4b )
```

If INITFFT is set to true, the call to INIT_FFT is done inside subroutine SYMLIN_FILTER2 and a call to END_FFT is also done before leaving subroutine SYMLIN_FILTER2. This optional argument has an effect only if argument USEFFT is used with the value true.

The default is INITFFT=true .

Further Details

No time observations will be lost, however the first and last (size(COEF)-1)/2 time observations are affected by end effects.

If USEFFT is used with the value true, the values at both ends of the output multi-channel time series are computed by assuming that the input multi-channel series is part of a periodic sequence of period size(VEC). Otherwise, each end of the filtered multi-channel time series is estimated by truncated the symmetric linear filter coefficients array.

The symmetric linear filter coefficients (e.g. the array COEF) can be computed with the help of functions LP_COEF, LP_COEF2, HP_COEF, HP_COEF2, BD_COEF and BD_COEF2.

6.27.30 `subroutine dan_filter ( vec, nsmooth, sym, trend )`

Purpose

Subroutine DAN_FILTER smooths an input time series (e.g. the argument VEC) by applying a Daniell filter (e.g. a simple moving average) of length NSMOOTH.

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On input, the vector containing the time series to be filtered. On output, the filtered time series is returned.

Size(VEC) must be greater or equal to 4.

NSMOOTH (INPUT) integer(i4b) The length of the Daniell filter to be applied to the time series. NSMOOTH must be odd. Moreover, NSMOOTH must be greater or equal to 3 and less or equal to size(VEC).

SYM (INPUT, OPTIONAL) real(stnd) An optional indictor variable used to designate wether the series has an even symmetry (SYM = one), an odd symmetry (SYM = -one) or no symmetry (SYM = zero). Other values than -one, one or zero are not allowed for the optional argument SYM.

The default value for SYM is one.

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+/-1 The mean of the time series is removed before time filtering
- TREND=+/-2 The drift from the time series is removed before time filtering by using the formula: drift = (VEC(size(VEC)) - VEC(1))/(size(VEC) - 1)
- TREND=+/-3 The least-squares line from the time series is removed before time filtering.

IF TREND=-1,-2 or -3, the mean, drift or least-squares line is reintroduced post-filtering, respectively. For other values of TREND nothing is done before or after filtering.

Further Details

Subroutine DAN_FILTER smooths an input time series by applying a Daniell filter as discussed in chapter 7 of Bloomfield (1976).

This subroutine use the hypothesis of the (even or odd) symmetry of the input time series to avoid losing values from the ends of the series.

For more details and algorithm, see:

(1) **Bloomfield, P.,1976:** Fourier analysis of time series- An introduction, John Wiley and Sons, New York, Chapter 7.

6.27.31 `subroutine dan_filter ( mat, nsmooth, sym, trend )`

Purpose

Subroutine DAN_FILTER smooths an input multi-channel time series (the argument MAT) by applying a Daniell filter (e.g. a simple moving average) of length NSMOOTH.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) The multi-channel time series matrix to be filtered. Each column of MAT corresponds to one observation. On output, the multi-channel filtered time series are returned.

Size(MAT,2) must be greater or equal to 4.

NSMOOTH (INPUT) integer(i4b) The length of the Daniell filter to be applied to the time series. NSMOOTH must be odd. Moreover, NSMOOTH must be greater or equal to 3 and less or equal to size(MAT,2).

SYM (INPUT, OPTIONAL) real(stnd) An optional indictor variable used to designate wether the series has an even symmetry (SYM = one), an odd symmetry (SYM = -one) or no symmetry (SYM = zero). Other values than -one, one or zero are not allowed for the optional argument SYM.

The default value for SYM is one.

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+/-1 The means of the time series are removed before time filtering
- TREND=+/-2 The drifts from the time series are removed before time filtering by using the formula: drift(:) = (MAT(:,size(MAT,2)) - MAT(:,1))/(size(MAT,2) - 1)
- TREND=+/-3 The least-squares lines from the time series are removed before time filtering.

IF TREND=-1,-2 or -3, the means, drifts or least-squares lines are reintroduced post-filtering, respectively. For other values of TREND nothing is done before or after filtering.

Further Details

Subroutine DAN_FILTER smooths an input multi-channel time series by applying a Daniell filter as discussed in chapter 7 of Bloomfield (1976).

This subroutine may use the hypothesis of the (even or odd) symmetry of the input time series to avoid losing values from the ends of the series.

For more details and algorithm, see

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York, Chapter 7.

6.27.32 subroutine moddan_filter (vec, smooth_param, sym, trend)

Purpose

Subroutine MODDAN_FILTER smooths an input time series (e.g. the argument VEC) by applying a sequence of modified Daniell filters.

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On input, the vector containing the time series to be filtered. On output, the filtered time series is returned. Size(VEC) must be greater or equal to 4.

SMOOTH_PARAM (INPUT) integer(i4b), dimension(:) The array of the half-lengths of the modified Daniell filters to be applied to the time series. All the values in SMOOTH_PARAM(:) must be greater than 0 and less than size(VEC) .

Size(SMOOTH_PARAM) must be greater or equal to 1.

SYM (INPUT, OPTIONAL) real(stnd) An optional indictor variable used to designate wether the series has an even symmetry (SYM = one), an odd symmetry (SYM = -one) or no symmetry (SYM = zero). Other values than -one, one or zero are not allowed for the optional argument SYM.

The default value for SYM is one.

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+/-1 The mean of the time series is removed before time filtering
- TREND=+/-2 The drift from the time series is removed before time filtering by using the formula: drift = (VEC(size(VEC)) - VEC(1))/(size(VEC) - 1)
- TREND=+/-3 The least-squares line from the time series is removed before time filtering.

IF TREND=-1,-2 or -3, the mean, drift or least-squares line is reintroduced post-filtering, respectively. For other values of TREND nothing is done before or after filtering.

Further Details

Subroutine MODDAN_FILTER smooths an input time series by applying a sequence of modified Daniell filters as discussed in chapter 7 of Bloomfield (1976). This subroutine use the hypothesis of the (even or odd) symmetry of the input time series to avoid losing values from the ends of the series.

For more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York, Chapter 7.

6.27.33 subroutine moddan_filter (mat, smooth_param, sym, trend)

Purpose

Subroutine MODDAN_FILTER smooths an input multi-channel time series (the argument MAT) by applying a sequence of modified Daniell filters.

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) The multi-channel time series matrix to be filtered. Each column of MAT corresponds to one observation. On output, the multi-channel filtered time series are returned.

Size(MAT,2) must be greater or equal to 4.

SMOOTH_PARAM (INPUT) integer(i4b), dimension(:) The array of the half-lengths of the modified Daniell filters to be applied to the time series. All the values in SMOOTH_PARAM(:) must be greater than 0 and less than size(MAT,2) .

Size(SMOOTH_PARAM) must be greater or equal to 1.

SYM (INPUT, OPTIONAL) real(stnd) An optional indictor variable used to designate wether the series has an even symmetry (SYM = one), an odd symmetry (SYM = -one) or no symmetry (SYM = zero). Other values than -one, one or zero are not allowed for the optional argument SYM.

The default value for SYM is one.

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+/-1 The means of the time series are removed before time filtering
- TREND=+/-2 The drifts from the time series are removed before time filtering by using the formula: drift(:) = (MAT(:,size(MAT,2)) - MAT(:,1))/(size(MAT,2) - 1)
- TREND=+/-3 The least-squares lines from the time series are removed before time filtering.

IF TREND=-1,-2 or -3, the means, drifts or least-squares lines are reintroduced post-filtering, respectively. For other values of TREND nothing is done before or after filtering.

Further Details

Subroutine MODDAN_FILTER smooths an input multi-channel time series by applying a sequence of modified Daniell filters as discussed in chapter 7 of Bloomfield (1976). This subroutine may use the hypothesis of the (even or odd) symmetry of the input time series to avoid losing values from the ends of the series.

For more details and algorithm see

(1) **Bloomfield, P., 1976: Fourier analysis of time series- An introduction,** John Wiley and Sons, New York, Chapter 7.

6.27.34 `function extend ( vec, index, sym )`

Purpose

This function returns the INDEX-th term in the series VEC, extending it if necessary with even or odd symmetry according to the sign of SYM, which should be either plus or minus one. (Note: the value zero will result in the extended value being zero).

Arguments

VEC (INPUT) real(stnd), dimension(:) On input, the vector containing the time series. If size(VEC) is zero, the extended value returned is zero.

INDEX (INPUT) integer(i4b) On input, the index of the desired term in the time series. INDEX may be any integer.

SYM (INPUT) real(stnd) An indictor variable used to designate wether the series has an even symmetry (SYM = one), an odd symmetry (SYM = -one) or no symmetry (SYM = zero). Other values than -one, one or zero are not allowed, however no checking is done on the SYM argument.

Further Details

For more details and algorithm, see

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York, Chapter 6.

6.27.35 `function extend ( mat, index, sym )`

Purpose

This function returns the INDEX-th term in the multi-channel series MAT, extending it if necessary with even or odd symmetry according to the sign of SYM, which should be either plus or minus one. Note: the value zero will result in the extended value being zero.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On input, the matrix containing the multi-channel time series. Each column of MAT corresponds to one observation. If size(MAT,2) is zero, the extended vector (which is dimensionned as size(MAT,1)) returned is zero.

INDEX (INPUT) integer(i4b) On input, the index of the desired term in the multi-channel time series.

SYM (INPUT) real(stnd) An indictor variable used to designate wether the series has an even symmetry (SYM = one), an odd symmetry (SYM = -one) or no symmetry (SYM = zero). Other values than -one, one or zero are not allowed, however no checking is done on the SYM argument.

Further Details

For more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York, Chapter 6.

6.27.36 `subroutine taper ( vec, taperp )`

Purpose

Subroutine TAPER applies a split-cosine-bell taper on an input time series (e.g. the argument VEC).

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On input, the vector containing the time series to be tapered. On output, the tapered time series is returned.

TAPERP (INPUT) real(stnd) The total percentage of the data to be tapered. TAPERP must be greater than zero and less or equal to one, otherwise the series is not tapered.

Further Details

This subroutine is adapted from Bloomfield (1976).

For more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York, Chapter 5.

6.27.37 `subroutine taper ( mat, taperp )`

Purpose

Subroutine TAPER applies a split-cosine-bell taper on an input multi-channel time series (the argument MAT).

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On input, the matrix containing the multi-channel time series. Each column of MAT corresponds to one observation.

TAPERP (INPUT) real(stnd) The total percentage of the data to be tapered. TAPERP must be greater than zero and less or equal to one, otherwise the series is not tapered.

Further Details

This subroutine is adapted from Bloomfield (1976).

For more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York, Chapter 5.

6.27.38 `function data_window ( n, win, taperp )`

Purpose

Function DATA_WINDOW computes data windows used in spectral computations.

Arguments

N (INPUT) integer(i4b) The size of the data window. N must be an even positive integer.

WIN (INPUT) integer(i4b) On entry, this argument specify the form of the data window. If:

- WIN=+1 The Bartlett window is used
- WIN=+2 The square window is used
- WIN=+3 The Welch window is used
- WIN=+4 The Hann window is used
- WIN=+5 The Hamming window is used
- WIN=+6 A split-cosine-bell window is used

For other values of WIN, a square window is returned.

TAPERP (INPUT, OPTIONAL) real(stnd) The total percentage of the data to be tapered if WIN=6. TAPERP must be greater than zero and less or equal to one, otherwise the default value is used.

The default is 0.2 .

Further Details

For more details, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York, Chapter 5.

6.27.39 function estim_dof (wk, win, smooth_param, l0, nseg, overlap)

Purpose

Function ESTIM_DOF computes "the equivalent number of degrees of freedom" of power and cross spectrum estimates as calculated by subroutines POWER_SPECTRUM, CROSS_SPECTRUM, POWER_SPECTRUM2 and CROSS_SPECTRUM2.

Arguments

WK (INPUT) real(stnd), dimension(:) On entry, this argument specify the data window used in the computations of the power and/or cross spectra.

Spectral computations are at (Size(WK)/2)+1 frequencies if the optional argument L0 is absent and are at ((Size(WK)+L0)/2)+1 frequencies if L0 is present (L0 is the number of zeros added to each segment).

Size(WK) must be greater or equal to 4 and Size(WK)+L0 must be even.

WIN (INPUT, OPTIONAL) integer(i4b) On entry, this argument specify the form of the data window given in argument WK. If:

- WIN=+1 The Bartlett window is used
- WIN=+2 The square window is used
- WIN=+3 The Welch window is used

- WIN=+4 The Hann window is used
- WIN=+5 The Hamming window is used
- WIN=+6 A split-cosine-bell window is used

For other values of WIN, a message error is issued and the program is stopped.

The default is WIN=+3, e.g. the Welch window.

SMOOTH_PARAM (INPUT, OPTIONAL) integer(i4b), dimension(:) if SMOOTH_PARAM is used, the power and/or cross spectrum have been estimated by repeated smoothing of the periodogram with modified Daniell weights.

On entry, SMOOTH_PARAM(:) gives the array of the half-lengths of the modified Daniell filters that have been applied.

All the values in SMOOTH_PARAM(:) must be greater than 0 and less than ((size(WK)+L0)/2) + 1 .

Size(SMOOTH_PARAM) must be greater or equal to 1.

L0 (INPUT, OPTIONAL) integer(i4b) The number of zeros added to the time series (or segment) in order to obtain more finely spaced spectral estimates. L0 must be a positive integer. Moreover, Size(VEC)+L0 must be even.

The default is L0=0, e.g. no zeros are added to the time series.

NSEG (INPUT, OPTIONAL) integer(i4b) The number of segments if the spectra have been computed by POWER_SPECTRUM2 and CROSS_SPECTRUM2 . NSEG must be a positive integer.

The segments are assumed to be independent or to overlap by one half of their length if the optional argument OVERLAP is used and is set to true. Let L = size(WK). Then, the number of segments may be computed as follows:

- N/L if OVERLAP=false
- (2N/L)-1 if OVERLAP=true

where N is equal to:

- the length of the original time series (call it M) if this length is evenly divisible by L,
- M+L-mod(M,L) if M is not evenly divisible L.

The default is NSEG=1, e.g. the time series is not segmented.

OVERLAP (INPUT, OPTIONAL) logical(lgl) If OVERLAP is set to false, the spectrum estimates have been computed from nonoverlapping segments.

If OVERLAP is set to true, the spectrum estimates have been computed from overlapped segments (subroutines POWER_SPECTRUM2 and CROSS_SPECTRUM2 may overlap the segments by one half of their length.

The default is OVERLAP=false .

Further Details

The computed equivalent number of degrees of freedom must be divided by two for the zero and Nyquist frequencies.

Furthermore, the computed equivalent number of degrees of freedom is not right near the zero and Nyquist frequencies if the PSD estimates have been smoothed by modified Daniell filters. The reason is that ESTIM_DOF assumes that smoothing involves averaging independent frequency ordinates. This is true

except near the zero and Nyquist frequencies where an average may contain contributions from negative frequencies, which are identical to and hence not independent of positive frequency spectral values. Thus, the number of degrees of freedom in PSD estimates near the 0 and Nyquist frequencies are as little as half the number of degrees of freedom of the spectral estimates away from these frequency extremes if the optional argument SMOOTH_PARAM is used.

For more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York, Chapter 8.

(2) **Welch, P.D., 1967:** The use of Fast Fourier Transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. IEEE trans. on audio and electroacoustics, Vol. Au-15, 2, 70-73.

6.27.40 function estim_dof2 (wk, l0, win, nsmooth, nseg, overlap)

Purpose

Function ESTIM_DOF2 computes "the equivalent number of degrees of freedom" of power and cross spectrum estimates as calculated by subroutines POWER_SPCTRM, CROSS_SPCTRM, POWER_SPCTRM2 and CROSS_SPCTRM2.

Arguments

WK (INPUT) real(stnd), dimension(:) On entry, this argument specifies the data window used in the computations of the power and/or cross spectra.

Spectral computations are at ((Size(WK)+L0)/2)+1 frequencies (L0 is the number of zeros added to each segment).

Size(WK) must be greater or equal to 4 and Size(WK)+L0 must be even.

L0 (INPUT) integer(i4b) The number of zeros added to the time series (or segment) in order to obtain more finely spaced spectral estimates. L0 must be a positive integer. Moreover, Size(VEC)+L0 must be even.

WIN (INPUT, OPTIONAL) integer(i4b) On entry, this argument specify the form of the data window given in argument WK. If:

- WIN=+1 The Bartlett window is used
- WIN=+2 The square window is used
- WIN=+3 The Welch window is used
- WIN=+4 The Hann window is used
- WIN=+5 The Hamming window is used
- WIN=+6 A split-cosine-bell window is used

For other values of WIN, a message error is issued and the program is stopped.

The default is WIN=+3, e.g. the Welch window.

NSMOOTH (INPUT, OPTIONAL) integer(i4b) if NSMOOTH is used, the power and/or cross spectra have been estimated by smoothing the periodogram with Daniell weights.

On entry, NSMOOTH gives the length of the Daniell filter that has been applied.

Setting NSMOOTH=0 on entry is equivalent to omit the optional argument NSMOOTH. Otherwise, NSMOOTH must be odd, greater than 2 and less or equal to ((size(WK)+L0)/2) + 1 .

NSEG (INPUT, OPTIONAL) integer(i4b) The number of segments if the spectra have been computed by POWER_SPCTRM2 and CROSS_SPCTRM2 . NSEG must be a positive integer.

The segments are assumed to be independent or to overlap by one half of their length if the optional argument OVERLAP is used and is set to true. Let L = size(WK). Then, the number of segments may be computed as follows:

- N/L if OVERLAP=false
- (2N/L)-1 if OVERLAP=true

where N is equal to:

- the length of the original time series (call it M) if this length is evenly divisible by L,
- M+L-mod(M,L) if M is not evenly divisible L.

The default is NSEG=1, e.g. the time series is not segmented.

OVERLAP (INPUT, OPTIONAL) logical(lgl) If OVERLAP is set to false, the spectrum estimates have been computed from nonoverlapping segments.

If OVERLAP is set to true, the spectrum estimates have been computed from overlapped segments (subroutines POWER_SPCTRUM2 and CROSS_SPCTRUM2 may overlap the segments by one half of their length.

The default is OVERLAP=false .

Further Details

For more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York, Chapter 8.

(2) **Welch, P.D., 1967:** The use of Fast Fourier Transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. IEEE trans. on audio and electroacoustics, Vol. Au-15, 2, 70-73.

6.27.41 subroutine comp_conflim (edof, probtest, conlwr, conupr, testcoher)

Purpose

Subroutine COMP_CONFLIM estimates confidence limit factors for spectral estimates and, optionally, critical value for testing the null hypothesis that squared coherency is zero.

Arguments

EDOF (INPUT) real(stnd) On entry, the equivalent number of degrees of freedom of the power spectrum estimates.

PROBTEST (INPUT, OPTIONAL) real(stnd) On entry, a probability. PROBTEST is the critical probability which is used to determine the lower and upper confidence limit factors (e.g. the optional arguments CONLWR and CONUPR) and the critical value for testing the null hypothesis that the

squared coherency is zero (e.g. the TESTCOHER optional argument). PROBTEST must verify 0. < P < 1.

The default is 0.05 .

CONLWR (OUTPUT, OPTIONAL) real(stnd)

CONUPR (OUTPUT, OPTIONAL) real(stnd) On output, these arguments specify the lower and upper (1-PROBTEST) * 100% confidence limit factors, respectively. Multiply the PSD estimates by these constants to get the lower and upper limits of a (1-PROBTEST) * 100% confidence interval for the PSD estimates.

TESTCOHER (OUTPUT, OPTIONAL) real(stnd) On output, this argument specifies the critical value for testing the null hypothesis that the squared coherency is zero at the PROBTEST * 100% significance level (e.g. squared coherencies less than TESTCOHER should be regarded as not significantly different from zero at the PROBTEST * 100% significance level).

6.27.42 subroutine comp_conflim (edof, probtest, conlwr, conupr, testcoher)

Purpose

Subroutine COMP_CONFLIM estimates confidence limit factors for spectral estimates and, optionally, critical values for testing the null hypothesis that squared coherencies are zero.

Arguments

EDOF (INPUT) real(stnd), dimension(:) On entry, the equivalent number of degrees of freedom of the power spectrum estimates.

PROBTEST (INPUT, OPTIONAL) real(stnd) On entry, a probability. PROBTEST is the critical probability which is used to determine the lower and upper confidence limit factors (e.g. the optional arguments CONLWR and CONUPR) and the critical value for testing the null hypothesis that the squared coherency is zero (e.g. the TESTCOHER optional argument). PROBTEST must verify 0. < P < 1.

The default is 0.05 .

CONLWR (OUTPUT, OPTIONAL) real(stnd), dimension(:)

CONUPR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On output, these arguments specify the lower and upper (1-PROBTEST) * 100% confidence limit factors, respectively. Multiply the PSD estimates by these constants to get the lower and upper limits of a (1-PROBTEST) * 100% confidence interval for the PSD estimates.

CONLWR must verify: size(CONLWR) = size(EDOF) .

CONUPR must verify: size(CONUPR) = size(EDOF) .

TESTCOHER (OUTPUT, OPTIONAL) real(stnd), dimension(:) On output, this argument specifies the critical values for testing the null hypothesis that the squared coherencies are zero at the PROBTEST * 100% significance level (e.g. squared coherencies less than TESTCOHER(:) should be regarded as not significantly different from zero at the PROBTEST * 100% significance level).

TESTCOHER must verify: size(TESTCOHER) = size(EDOF) .

6.27.43 subroutine spctrm_ratio (edofn, edofd, lwr_ratio, upr_ratio, pinterval)

Purpose

Subroutine SPCTRM_RATIO calculates a pointwise tolerance interval for the ratio of two estimated spectra under the assumption that the two "true" underlying spectra are the same.

Arguments

EDOFN (INPUT) real(stnd) On exit, the equivalent number of degrees of freedom of the first estimated spectrum (e.g. the numerator of the ratio of the two estimated spectra).

EDOFN must be greater than zero.

EDOFD (INPUT) real(stnd) On exit, the equivalent number of degrees of freedom of the second estimated spectrum (e.g. the denominator of the ratio of the two estimated spectra).

EDOFD must be greater than zero.

LWR_RATIO (OUTPUT) real(stnd)

UPR_RATIO (OUTPUT) real(stnd) On output, these arguments specify the lower and upper critical ratios of the computed PINTERVAL * 100% tolerance interval for the ratio of the power spectral density estimates.

PINTERVAL (INPUT, OPTIONAL) real(stnd) On entry, a probability. This probability is used to determine the upper and lower critical ratios of the computed tolerance interval. A PINTERVAL * 100% tolerance interval is computed and output in the two arguments LWR_RATIO and UPR_RATIO. PINTERVAL must verify: 0. < PINTERVAL < 1.

The default value is 0.90, e.g. a 90% tolerance interval is computed.

Further Details

For more details, see:

(1) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford, Chapter 4.

6.27.44 subroutine spctrm_ratio (edofn, edofd, lwr_ratio, upr_ratio, pinterval)

Purpose

Subroutine SPCTRM_RATIO calculates pointwise tolerance intervals for the ratio of two estimated spectra under the assumption that the two "true" underlying spectra are the same.

Arguments

EDOFN (INPUT) real(stnd), dimension(:) On exit, the equivalent number of degrees of freedom of the first estimated spectrum (e.g. the numerator of the ratio of the two estimated spectra).

Elements of EDOFN(:) must be greater than zero.

EDOFD (INPUT) real(stnd), dimension(:) On exit, the equivalent number of degrees of freedom of the second estimated spectrum (e.g. the denominator of the ratio of the two estimated spectra). Elements of EDOFD(:) must be greater than zero.

EDOFD must verify: size(EDOFD) = size(EDOFN) .

LWR_RATIO (OUTPUT) real(stnd), dimension(:)

UPR_RATIO (OUTPUT) real(stnd), dimension(:) On output, these arguments specify the lower and upper critical ratios of the computed PINTERVAL * 100% tolerance interval for the ratio of the power spectral density estimates.

LWR_RATIO must verify: size(LWR_RATIO) = size(EDOFN) .

UPR_RATIO must verify: size(UPR_RATIO) = size(EDOFN) .

PINTERVAL (INPUT, OPTIONAL) real(stnd) On entry, a probability. This probability is used to determine the upper and lower critical ratios of the computed tolerance interval. A PINTERVAL * 100% tolerance interval is computed and output in the two arguments LWR_RATIO and UPR_RATIO. PINTERVAL must verify: 0. < PINTERVAL < 1.

The default value is 0.90, e.g. a 90% tolerance interval is computed .

Further Details

For more details, see:

(1) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford, Chapter 4.

6.27.45 subroutine spctrm_ratio2 (psvecn, psvecd, edofn, edofd, prob, min_ratio, max_ratio, prob_min_ratio, prob_max_ratio)

Purpose

Subroutine SPCTRM_RATIO2 calculates a conservative critical probability value (e.g. p-value) for testing the hypothesis of a common spectrum for two estimated spectra (e.g. the arguments PSVECN, PSVECD). This conservative critical probability value is computed from the minimim and maximum values of the ratio of the two estimated spectra and the associated probabilities of obtaining, respectively, a value less (for the minimum ratio) and higher (for the maximum ratio) than attained under the null hypothesis of a common spectrum for the two time series.

Arguments

PSVECN (INPUT) real(stnd), dimension(:) On entry, a real vector containing the Power Spectral Density (PSD) estimates of the first time series (e.g. the numerator of the ratio of the two estimated spectra).

All elements in PSVECN(:) must be greater or equal to zero and size(PSVECN) must be greater or equal to 2.

PSVECD (INPUT) real(stnd), dimension(:) On entry, a real vector containing the Power Spectral Density (PSD) estimates of the second time series (e.g. the denominator of the ratio of the two estimated spectra). All elements in PSVECD(:) must be greater than zero and size(PSVECD) must be greater or equal to 2.

PSVECD must also verify: size(PSVECD) = size(PSVECN) .

EDOFN (INPUT) real(stnd) On exit, the equivalent number of degrees of freedom of the first estimated spectrum (e.g. the numerator of the ratio of the two estimated spectra).

EDOFN must be greater than zero.

EDOFD (INPUT) real(stnd) On exit, the equivalent number of degrees of freedom of the second estimated spectrum (e.g. the denominator of the ratio of the two estimated spectra).

EDOFD must be greater than zero.

PROB (OUTPUT) real(stnd) On exit, the conservative critical probability value (e.g. p-value) computed under the hypothesis that the two "true" underlying spectra are the same. See the description of the PROB_MIN_RATIO and PROB_MAX_RATIO optional arguments for more details.

MIN_RATIO (OUTPUT, OPTIONAL) real(stnd)

MAX_RATIO (OUTPUT, OPTIONAL) real(stnd) On output, these arguments give, respectively, the minimum and maximum values of the ratio of the two PSD estimates.

PROB_MIN_RATIO (OUTPUT, OPTIONAL) real(stnd)

PROB_MAX_RATIO (OUTPUT, OPTIONAL) real(stnd) On output, these arguments give, respectively, the probabilities of obtaining a smaller value of the minimum ratio (e.g. the argument MIN_RATIO) and a greater value of the maximum ratio (e.g. the argument MAX_RATIO) under the null hypothesis that the two "true" underlying spectra are the same.

The PROB argument is computed as 2 * min(PROB_MIN_RATIO,PROB_MAX_RATIO).

Further Details

This statistical test relies on the assumptions that the different spectral ordinates have the same sampling distribution and are independent of each other for each series. This means, in particular, that the spectral ordinates corresponding to the zero and Nyquist frequencies must be excluded from the PSVECN and PSVECD vectors before calling SPCTRM_RATIO2 and that the two estimated spectra have not been obtained by smoothing the periodogram in the frequency domain.

It is also assumed that the PSVECN and PSVECD realizations are independent.

For more details, see:

(1) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford, Chapter 4.

6.27.46 subroutine spctrm_ratio2 (psmatn, psmatd, edofn, edofd, prob, min_ratio, max_ratio, prob_min_ratio, prob_max_ratio)

Purpose

Subroutine SPCTRM_RATIO2 calculates conservative critical probability values (e.g. p-values) for testing the hypothesis of a common spectrum for the elements of two estimated multi-channel spectra (e.g. the arguments PSMATN, PSMATD).

These conservative critical probability values are computed from the minimim and maximum values of the ratio of the two estimated multi-channel spectra and the associated probabilities of obtaining, respectively, a value less (for the minimum ratio) and higher (for the maximum ratio) than attained under the null hypothesis of a common spectrum for the two multi-channel time series.

Arguments

PSMATN (INPUT) real(stnd), dimension(:,:) On entry, a real matrix containing the Power Spectral Density (PSD) estimates of the first multi-channel time series (e.g. the numerator of the ratio of the two estimated multi-channel spectra). Each row of the real matrix PSMATN contains the estimated spectrum of the corresponding "row" of the first multi-channel times series.

All elements in PSMATN(:,:) must be greater or equal to zero and size(PSMATN,2) must be greater or equal to 2.

PSMATD (INPUT) real(stnd), dimension(:,:) On entry, a real matrix containing the Power Spectral Density (PSD) estimates of the second multi-channel time series (e.g. the denominator of the ratio of the two estimated multi-channel spectra). Each row of the real matrix PSMATD contains the estimated spectrum of the corresponding "row" of the second multi-channel times series. All elements in PSMATD(:,:) must be greater than zero and size(PSMATD,2) must be greater or equal to 2.

PSMATD must also verify:

- size(PSMATD,1) = size(PSMATN,1) ,
- size(PSMATD,2) = size(PSMATN,2) .

EDOFN (INPUT) real(stnd) On exit, the equivalent number of degrees of freedom of the first estimated spectrum (e.g. the numerator of the ratio of the two estimated spectra).

EDOFN must be greater than zero.

EDOFD (INPUT) real(stnd) On exit, the equivalent number of degrees of freedom of the second estimated spectrum (e.g. the denominator of the ratio of the two estimated spectra).

EDOFD must be greater than zero.

PROB (OUTPUT) real(stnd), dimension(:) On exit, the conservative critical probability values (e.g. p-values) computed under the hypothesis that the two "true" underlying multi-channel spectra are the same. See the description of the PROB_MIN_RATIO and PROB_MAX_RATIO optional arguments for more details.

PROB must verify: size(PROB) = size(PSMATN,1) .

MIN_RATIO (OUTPUT, OPTIONAL) real(stnd), dimension(:)

MAX_RATIO (OUTPUT, OPTIONAL) real(stnd), dimension(:) On output, these arguments give, respectively, the minimum and maximum values of the ratio of the two multi-channel PSD estimates.

MIN_RATIO must verify: size(MIN_RATIO) = size(PSMATN,1) .

MAX_RATIO must verify: size(MAX_RATIO) = size(PSMATN,1) .

PROB_MIN_RATIO (OUTPUT, OPTIONAL) real(stnd), dimension(:)

PROB_MAX_RATIO (OUTPUT, OPTIONAL) real(stnd), dimension(:) On output, these arguments give, respectively, the probabilities of obtaining a smaller value of the minimum ratio (e.g. the argument MIN_RATIO) and a greater value of the maximum ratio (e.g. the argument MAX_RATIO) under the null hypothesis that the two "true" underlying multi-channel spectra are the same. The PROB(:) argument is calculated as 2 * min(PROB_MIN_RATIO(:),PROB_MAX_RATIO(:)).

PROB_MIN_RATIO must verify: size(PROB_MIN_RATIO) = size(PSMATN,1) .

PROB_MAX_RATIO must verify: size(PROB_MAX_RATIO) = size(PSMATN,1) .

Further Details

This statistical test relies on the assumptions that the different spectral ordinates have the same sampling distribution and are independent of each other for each series. This means, in particular, that the spectral ordinates corresponding to the zero and Nyquist frequencies must be excluded from the PSMATN and PSMATD matrices before calling SPCTRM_RATIO2 and that the two estimated multi-channel spectra have not been obtained by smoothing the periodogram in the frequency domain.

It is also assumed that the PSMATN and PSMATD realizations are independent.

For more details, see:

(1) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford, Chapter 4.

6.27.47 subroutine spctrm_ratio3 (psvecn, psvecd, edofn, edofd, chi2_stat, prob)

Purpose

Subroutine SPCTRM_RATIO3 calculates an approximate critical probability value (e.g. p-value) for testing the hypothesis of a common spectrum for two estimated spectra (e.g. the arguments PSVECN, PSVECD). This approximate critical probability value is derived from the following CHI2 statistic :

CHI2_STAT = (2/EDOFN + 2/EDOFD)**(-1) [sum k=1 to nf] log(PSVECN(k) / PSVECD(k))**(2)

where nf = size(PSVECN) = size(PSVECD). In order to derive an approximate critical probability value, it is assumed that CHI2_STAT has an approximate CHI2 distribution with nf degrees of freedom.

Arguments

PSVECN (INPUT) real(stnd), dimension(:) On entry, a real vector containing the Power Spectral Density (PSD) estimates of the first time series (e.g. the numerator of the ratio of the two estimated spectra).

All elements in PSVECN(:) must be greater than zero and size(PSVECN) must be greater or equal to 2.

PSVECD (INPUT) real(stnd), dimension(:) On entry, a real vector containing the Power Spectral Density (PSD) estimates of the second time series (e.g. the denominator of the ratio of the two estimated spectra).

All elements in PSVECD(:) must be greater than zero and size(PSVECD) must be greater or equal to 2.

PSVECD must verify: size(PSVECD) = size(PSVECN) .

EDOFN (INPUT) real(stnd) On exit, the equivalent number of degrees of freedom of the first estimated spectrum (e.g. the numerator of the ratio of the two estimated spectra).

EDOFN must be greater than one.

EDOFD (INPUT) real(stnd) On exit, the equivalent number of degrees of freedom of the second estimated spectrum (e.g. the denominator of the ratio of the two estimated spectra).

EDOFD must be greater than one.

CHI2_STAT (OUTPUT) real(stnd) On output, the CHI2 statistic which is assumed to follow a CHI2 distribution with size(PSVECN) degrees of freedom under the null hypothesis of a common spectrum.

PROB (OUTPUT) real(stnd) On exit, the aproximate critical probability value (e.g. p-value) computed under the hypothesis that the two "true" underlying spectra are the same. PROB is calculated as the probability of obtaining a value greater or equal to CHI2_STAT under the hypothesis of a common spectrum for the two series.

Further Details

This statistical test relies on the assumptions that the different spectral ordinates have the same sampling distribution and are independent of each other for each time series. This means, in particular, that the spectral ordinates corresponding to the zero and Nyquist frequencies must be excluded from the PSVECN and PSVECD vectors before calling SPCTRM_RATIO3 and that the two estimated spectra have not been obtained by smoothing the periodogram in the frequency domain.

It is also assumed that the PSVECN and PSVECD realizations are independent.

6.27.48 subroutine spctrm_ratio3 (psmatn, psmatd, edofn, edofd, chi2_stat, prob)

Purpose

Subroutine SPCTRM_RATIO3 calculates approximate critical probability values (e.g. p-values) for testing the hypothesis of a common spectrum for two estimated multi-channel spectra (eg the arguments PSMATN, PSMATD). Thess approximate critical probability values are derived from the following CHI2 statistics :

CHI2_STAT(:n) = (2/EDOFN + 2/EDOFD)**(-1) [sum k=1 to nf] log(PSMATN(:n,k) / PSMATD(:n,k))**(2)

where n = size(PSMATN,1) = size(PSMATD,1) = size(CHI2_STAT) and nf = size(PSMATN,2) = size(PSMATD,2). In order to derive approximate critical probability values, it is assumed that each element of CHI2_STAT(:n) has an approximate CHI2 distribution with nf degrees of freedom.

Arguments

PSMATN (INPUT) real(stnd), dimension(:,:) On entry, a real matrix containing the Power Spectral Density (PSD) estimates of the first multi-channel time series (e.g. the numerator of the ratio of the two estimated multi-channel spectra). Each row of the real matrix PSMATN contains the estimated spectrum of the corresponding "row" of the first multi-channel times series.

All elements in PSMATN(:,:) must be greater than zero and size(PSMATN,2) must be greater or equal to 2.

PSMATD (INPUT) real(stnd), dimension(:,:) On entry, a real matrix containing the Power Spectral Density (PSD) estimates of the second multi-channel time series (e.g. the denominator of the ratio of the two estimated multi-channel spectra). Each row of the real matrix PSMATD contains the estimated spectrum of the corresponding "row" of the second multi-channel times series. All elements in PSMATD(:,:) must be greater than zero and size(PSMATD,2) must be greater or equal to 2.

PSMATD must also verify:

- size(PSMATD,1) = size(PSMATN,1) ,

- size(PSMATD,2) = size(PSMATN,2) .

EDOFN (INPUT) real(stnd) On exit, the equivalent number of degrees of freedom of the first estimated spectrum (e.g. the numerator of the ratio of the two estimated spectra).

EDOFN must be greater than one.

EDOFD (INPUT) real(stnd) On exit, the equivalent number of degrees of freedom of the second estimated spectrum (e.g. the denominator of the ratio of the two estimated spectra).

EDOFD must be greater than one.

CHI2_STAT (OUTPUT) real(stnd), dimension(:) On output, the CHI2 statistics which are assumed to follow a CHI2 distribution with size(PSMATN,2) degrees of freedom under the null hypothesis of a common spectrum.

CHI2_STAT must verify: size(CHI2_STAT) = size(PSMATN,1) .

PROB (OUTPUT) real(stnd), dimension(:) On exit, the aproximate critical probability values (e.g. p-values) computed under the hypothesis that the two "true" underlying multi-channel spectra are the same. Each element of PROB(:) is calculated as the probability of obtaining a value greater or equal to the corresponding element of CHI2_STAT(:) under the hypothesis of a common spectrum for the two (single) series.

PROB must verify: size(PROB) = size(PSMATN,1) .

Further Details

This statistical test relies on the assumptions that the different spectral ordinates have the same sampling distribution and are independent of each other for each time series. This means, in particular, that the spectral ordinates corresponding to the zero and Nyquist frequencies must be excluded from the PSMATN and PSMATD matrices before calling SPCTRM_RATIO3 and that the two estimated multi-channel spectra have not been obtained by smoothing the periodogram in the frequency domain.

It is also assumed that the PSMATN and PSMATD realizations are independent.

6.27.49 subroutine spctrm_ratio4 (psvecn, psvecd, edofn, edofd, range_stat, prob)

Purpose

Subroutine SPCTRM_RATIO4 calculates an approximate critical probability value (e.g. p-value) for testing the hypothesis of a common shape for two estimated spectra (e.g. the arguments PSVECN, PSVECD). This approximate critical probability value is derived from the following RANGE statistic :

RANGE_STAT = (2/EDOFN + 2/EDOFD)**(-1/2) * (maxval(logratio(:nf)) - minval(logratio(:nf)))

where nf = size(PSVECN) = size(PSVECD) and logratio(:nf) is given as

logratio(:nf) = log(PSVECN(:nf) / PSVECD(:nf))

In order to derive an approximate critical probability value, it is assumed that the elements of the vector logratio(:nf) are independent and follow approximately a normal distribution with mean (1/EDOFN) - (1/EDOFD) and variance (2/EDOFN) + (2/EDOFD). Than, the distribution of the statistic RANGE_STAT may be approximated by the distribution function of the range of nf independent normal random variables with mean and variance as specified above.

Arguments

PSVECN (INPUT) real(stnd), dimension(:) On entry, a real vector containing the Power Spectral Density (PSD) estimates of the first time series (e.g. the numerator of the ratio of the two estimated spectra).

All elements in PSVECN(:) must be greater than zero and size(PSVECN) must be greater or equal to 2.

PSVECD (INPUT) real(stnd), dimension(:) On entry, a real vector containing the Power Spectral Density (PSD) estimates of the second time series (e.g. the denominator of the ratio of the two estimated spectra).

All elements in PSVECD(:) must be greater than zero and size(PSVECD) must be greater or equal to 2.

PSVECD must also verify: size(PSVECD) = size(PSVECN) .

EDOFN (INPUT) real(stnd) On exit, the equivalent number of degrees of freedom of the first estimated spectrum (e.g. the numerator of the ratio of the two estimated spectra).

EDOFN must be greater than one.

EDOFD (INPUT) real(stnd) On exit, the equivalent number of degrees of freedom of the second estimated spectrum (e.g. the denominator of the ratio of the two estimated spectra).

EDOFD must be greater than one.

RANGE_STAT (OUTPUT) real(stnd) On output, the range statistic which is assumed to follow the distribution of the range of nf=size(PSVECN) independent standard normal variates under the null hypothesis of a common shape spectrum.

PROB (OUTPUT) real(stnd) On exit, the aproximate critical probability value (e.g. p-value) computed under the hypothesis that the two "true" underlying spectra have the same shape. PROB is calculated as the probability of obtaining a value greater or equal to RANGE_STAT under the hypothesis of a common shape spectrum for the two series.

Further Details

This statistical test relies on the assumptions that the different spectral ordinates have the same sampling distribution and are independent of each other for each time series. This means, in particular, that the spectral ordinates corresponding to the zero and Nyquist frequencies must be excluded from the PSVECN and PSVECD vectors before calling SPCTRM_RATIO4 and that the two estimated spectra have not been obtained by smoothing the periodogram in the frequency domain.

It is also assumed that the PSVECN and PSVECD realizations are independent.

For more details, see:

(1) **Coates, D.S., and Diggle, P.J., 1986:** Tests for comparing two estimated spectral densities. J. Time series Analysis, Vol. 7, pp. 7-20 .

(2) **Potscher, B.,M., and Reschenhofer, E., 1988:** Discriminating between two spectral densities in case of replicated observations. J. Time series Analysis, Vol. 9, pp. 221-224 .

(3) **Potscher, B.,M., and Reschenhofer, E., 1989:** Distribution of the Coates-Diggle test statistic in case of replicated observations. Statistics, Vol. 20, pp. 417-421 .

6.27.50 subroutine spctrm_ratio4 (psmatn, psmatd, edofn, edofd, range_stat, prob)

Purpose

Subroutine SPCTRM_RATIO4 calculates approximate critical probability values (e.g. p-values) for testing the hypothesis of a common shape for two estimated multi-channel spectra (e.g. the arguments PSMATN, PSMATD). These approximate critical probability values are derived from the following range statistics :

RANGE_STAT(:n) = (2/EDOFN + 2/EDOFD)**(-1/2) * (maxval(logratio(:n,:nf), dim=2) - minval(logratio(:n,:nf), dim=2))

where n = size(PSMATN,1) = size(PSMATD,1), nf = size(PSMATN,2) = size(PSMATD,2) and logratio(:n,:nf) is given as

logratio(:n,:nf) = log(PSMATN(:n,:nf) / PSMATD(:n,:nf))

In order to derive approximate critical probability values, it is assumed that the elements of the vectors logratio(i,:nf), for i=1 to n, are independent and follow approximately a normal distribution with mean (1/EDOFN) - (1/EDOFD) and variance (2/EDOFN) + (2/EDOFD). Than, the distribution of the statistics RANGE_STAT(:n) may be approximated by the distribution function of the range of nf independent normal random variables with mean and variance as specified above.

Arguments

PSMATN (INPUT) real(stnd), dimension(:,:) On entry, a real matrix containing the Power Spectral Density (PSD) estimates of the first multi-channel time series (e.g. the numerator of the ratio of the two estimated multi-channel spectra). Each row of the real matrix PSMATN contains the estimated spectrum of the corresponding "row" of the first multi-channel times series.

All elements in PSMATN(:,:) must be greater than zero and size(PSMATN,2) must be greater or equal to 2.

PSMATD (INPUT) real(stnd), dimension(:,:) On entry, a real matrix containing the Power Spectral Density (PSD) estimates of the second multi-channel time series (e.g. the denominator of the ratio of the two estimated multi-channel spectra). Each row of the real matrix PSMATD contains the estimated spectrum of the corresponding "row" of the second multi-channel times series.

All elements in PSMATD(:,:) must be greater than zero and size(PSMATD,2) must be greater or equal to 2.

PSMATD must also verify:

- size(PSMATD,1) = size(PSMATN,1) ,
- size(PSMATD,2) = size(PSMATN,2) .

EDOFN (INPUT) real(stnd) On exit, the equivalent number of degrees of freedom of the first estimated spectrum (e.g. the numerator of the ratio of the two multi-channel estimated spectra).

EDOFN must be greater than one.

EDOFD (INPUT) real(stnd) On exit, the equivalent number of degrees of freedom of the second estimated spectrum (e.g. the denominator of the ratio of the two multi-channel estimated spectra).

EDOFD must be greater than one.

RANGE_STAT (OUTPUT) real(stnd), dimension(:) On output, the range statistics which are assumed to follow the distribution of the range of nf=size(PSMATN,2) independent standard normal variates under the null hypothesis of a common shape spectrum.

RANGE_STAT must verify: size(RANGE_STAT) = size(PSMATN,1) .

PROB (OUTPUT) real(stnd), dimension(:) On exit, the aproximate critical probability values (e.g. p-values) computed under the hypothesis that the two "true" underlying multi-channel spectra have the same shape.

Each element of PROB(:) is calculated as the probability of obtaining a value greater or equal to the corresponding element of RANGE_STAT(:) under the hypothesis of a common shape spectrum for the two multi-channel series.

Further Details

This statistical test relies on the assumptions that the different spectral ordinates have the same sampling distribution and are independent of each other for each multi-channel time series. This means, in particular, that the spectral ordinates corresponding to the zero and Nyquist frequencies must be excluded from the PSMATN and PSMATD matrices before calling SPCTRM_RATIO4 and that the two estimated multi-channel spectra have not been obtained by smoothing the periodogram in the frequency domain.

It is also assumed that the PSMATN and PSMATD multi-channel realizations are independent.

For more details, see:

(1) **Coates, D.S., and Diggle, P.J., 1986:** Tests for comparing two estimated spectral densities. J. Time series Analysis, Vol. 7, pp. 7-20 .

(2) **Potscher, B.,M., and Reschenhofer, E., 1988:** Discriminating between two spectral densities in case of replicated observations. J. Time series Analysis, Vol. 9, pp. 221-224 .

(3) **Potscher, B.,M., and Reschenhofer, E., 1989:** Distribution of the Coates-Diggle test statistic in case of replicated observations. Statistics, Vol. 20, pp. 417-421 .

6.27.51 subroutine spctrm_diff (psvec1, psvec2, ks_stat, prob, nrep, norm, initseed)

Purpose

Subroutine SPCTRM_DIFF calculates an approximate critical probability value (e.g. p-value) for testing the hypothesis of a common shape for two estimated spectra (e.g. the arguments PSVEC1 and PSVEC2). This approximate critical probability value is derived from the following Kolmogorov-Smirnov statistic (e.g. the KS_STAT argument) :

D = [sup m=1 to nf] | F1(m) - F2(m) |

where nf = size(PSVEC1) = size(PSVEC2), F1(:) and F2(:) are the normalized cumulative periodograms computed from the estimated spectra PSVEC1(:) and PSVEC2(:). The distribution of D under the null hypothesis of a common shape for the spectra of the two series is approximated by calculating D for some large number (e.g. the NREP argument) of random interchanges of periodogram ordinates at each frequency for the two estimated spectra (e.g. the arguments PSVEC1(:) and PSVEC2(:)).

Arguments

PSVEC1 (INPUT) real(stnd), dimension(:) On entry, a real vector containing the Power Spectral Density (PSD) estimates of the first time series.

size(PSVECN) must be greater or equal to 10.

PSVEC2 (INPUT) real(stnd), dimension(:) On entry, a real vector containing the Power Spectral Density (PSD) estimates of the second time series.

PSVEC2 must verify: size(PSVEC2) = size(PSVEC1) .

KS_STAT (OUTPUT) real(stnd) On output, the Kolmogorov-Smirnov statistic.

PROB (OUTPUT) real(stnd) On exit, the aproximate critical probability value (e.g. p-value) computed under the hypothesis that the two "true" underlying spectra have the same shape. PROB is calculated as the probability of obtaining a value greater or equal to KS_STAT under the hypothesis of a common shape for the spectra of the two series.

NREP (INPUT, OPTIONAL) integer(i4b) On entry, when argument NREP is present, NREP specifies the number of random interchanges of periodogram ordinates at each frequency in order to approximate the randomization distribution of KS_STAT under the null hypothesis.

The default is 99.

NORM (INPUT, OPTIONAL) logical(lgl) On entry, if:

- NORM=true, KS_STAT is calculated from normalized cumulative periodograms.
- NORM=false KS_STAT is calculated from non-normalized cumulative periodograms. In that case the null hypothesis is that the spectra of the two time series is the same.

The default is NORM=true.

INITSEED (INPUT, OPTIONAL) logical(lgl) On entry, if INITSEED=true, a call to RANDOM_SEED_() without arguments is done in the subroutine, in order to initiate a non-repeatable reset of the seed used by the STATPACK random generator.

The default is INITSEED=false.

Further Details

This statistical randomization test relies on the assumptions that the different spectral ordinates have the same sampling distribution and are independent of each other. This means, in particular, that the spectral ordinates corresponding to the zero and Nyquist frequencies must be excluded from the PSVEC1 and PSVEC2 vectors before calling SPCTRM_DIFF and that the two estimated spectra have not been obtained by smoothing the periodogram in the frequency domain.

This randomization test could only be used to compare two periodograms or two spectral estimates computed as the the average of, say, r periodograms for each time series.

For more details, see:

(1) **Diggle, P.J., and Fisher, N.I., 1991:** Nonparametric comparison of cumulative periodograms, Applied Statistics, Vol. 40, No 3, pp. 423-434.

6.27.52 subroutine spctrm_diff (psmat1, psmat2, ks_stat, prob, nrep, norm, initseed)

Purpose

Subroutine SPCTRM_DIFF calculates approximate critical probability values (e.g. p-value) for testing the hypothesis of a common shape for two estimated multi-channel spectra (e.g. the arguments PSMAT1 and PSMAT2). These approximate critical probability values are derived from the following Kolmogorov-Smirnov statistics (e.g. the KS_STAT(:) argument) :

D(j) = [sup m=1 to nf] | F1(j,m) - F2(j,m) | for j=1, . . . , size(PSMAT1,1)

where nf = size(PSMAT1,2) = size(PSMAT2,2), F1(:,:) and F2(:,:) are the normalized cumulative periodograms computed from the estimated spectra PSMAT1(:,:) and PSMAT2(:,:). The distribution of D under the null hypothesis of a common shape for the spectra of the two multi-channel series is approximated by calculating D for some large number (e.g. the NREP argument) of random interchanges of periodogram ordinates at each frequency for the two estimated multi-channel spectra (e.g. the arguments PSMAT1(:,:) and PSMAT2(:,:)).

Arguments

PSMAT1 (INPUT) real(stnd), dimension(:,:) On entry, a real matrix containing the Power Spectral Density (PSD) estimates of the first multi-channel time series. Each row of the real matrix PSMAT1 contains the estimated spectrum of the corresponding "row" of the first multi-channel times series.

size(PSMATN,2) must be greater or equal to 10.

PSMAT2 (INPUT) real(stnd), dimension(:,:) On entry, a real matrix containing the Power Spectral Density (PSD) estimates of the second multi-channel time series. Each row of the real matrix PSMAT2 contains the estimated spectrum of the corresponding "row" of the second multi-channel times series.

PSMAT2 must verify:

- size(PSMAT2,1) = size(PSMAT1,1) ,
- size(PSMAT2,2) = size(PSMAT1,2) .

KS_STAT (OUTPUT) real(stnd), dimension(:) On output, the Kolmogorov-Smirnov statistics for the multi-channel times series .

KS_STAT must verify: size(KS_STAT) = size(PSMAT1,1) .

PROB (OUTPUT) real(stnd), dimension(:) On exit, the aproximate critical probability values (e.g. p-values) computed under the hypothesis that the two "true" underlying multi-channel spectra have the same shape.

PROB is calculated as the probability of obtaining a value greater or equal to KS_STAT under the hypothesis of a common shape for the spectra of the two series.

PROB must verify: size(PROB) = size(PSMAT1,1) .

NREP (INPUT, OPTIONAL) integer(i4b) On entry, when argument NREP is present, NREP specifies the number of random interchanges of periodogram ordinates at each frequency in order to approximate the randomization distribution of KS_STAT under the null hypothesis.

The default is 99.

NORM (INPUT, OPTIONAL) logical(lgl) On entry, if:

- NORM=true, KS_STAT is calculated from normalized cumulative periodograms.
- NORM=false KS_STAT is calculated from non-normalized cumulative periodograms. In that case the null hypothesis is that the spectra of the two multi-channel time series is the same.

The default is NORM=true.

INITSEED (INPUT, OPTIONAL) logical(lgl) On entry, if INITSEED=true, a call to RANDOM_SEED_() without arguments is done in the subroutine, in order to initiate a non-repeatable reset of the seed used by the STATPACK random generator.

The default is INITSEED=false.

Further Details

This statistical randomization test relies on the assumptions that the different spectral ordinates have the same sampling distribution and are independent of each other. This means, in particular, that the spectral ordinates corresponding to the zero and Nyquist frequencies must be excluded from the PSMAT1 and PSMAT2 matrices before calling SPCTRM_DIFF and that the two estimated multi-channel spectra have not been obtained by smoothing the periodograms in the frequency domain.

This randomization test could only be used to compare two periodograms or two spectral estimates computed as the the average of, say, r periodograms for each time series.

For more details see:

(1) **Diggle, P.J., and Fisher, N.I., 1991:** Nonparametric comparison of cumulative periodograms, Applied Statistics, Vol. 40, No 3, pp. 423-434.

6.27.53 subroutine spctrm_diff2 (psvec1, psvec2, chi2_stat, prob, nrep, initseed)

Purpose

Subroutine SPCTRM_DIFF2 calculates an approximate critical probability value (e.g. p-value) for testing the hypothesis of a common underlying spectrum for the two estimated spectra (e.g. the arguments PSVEC1 and PSVEC2). This approximate critical probability value is derived from the following CHI2 statistic (e.g. the CHI2_STAT argument) :

CHI2_STAT = (1/nf) [sum k=1 to nf] log(PSVEC1(k) / PSVEC2(k))**(2)

where nf = size(PSVEC1) = size(PSVEC2). The distribution of CHI2_STAT under the null hypothesis of a common spectrum for the two series is approximated by calculating CHI2_STAT for some large number (e.g. the NREP argument) of random interchanges of periodogram ordinates at each frequency for the two estimated spectra (e.g. the arguments PSVEC1(:) and PSVEC2(:)).

Arguments

PSVEC1 (INPUT) real(stnd), dimension(:) On entry, a real vector containing the Power Spectral Density (PSD) estimates of the first time series.

All elements in PSVEC1(:) must be greater than zero. size(PSVECN) must be greater or equal to 10.

PSVEC2 (INPUT) real(stnd), dimension(:) On entry, a real vector containing the Power Spectral Density (PSD) estimates of the second time series.

All elements in PSVEC2(:) must be greater than zero.

PSVEC2 must verify: size(PSVEC2) = size(PSVEC1) .

CHI2_STAT (OUTPUT) real(stnd) On output, the CHI2 statistic.

PROB (OUTPUT) real(stnd) On exit, the aproximate critical probability value (e.g. p-value) computed under the hypothesis that the two "true" underlying spectra are the same.

PROB is calculated as the probability of obtaining a value greater or equal to CHI2_STAT under the hypothesis of a common spectrum for the two series.

NREP (INPUT, OPTIONAL) integer(i4b) On entry, when argument NREP is present, NREP specifies the number of random interchanges of periodogram ordinates at each frequency in order to approximate the randomization distribution of CHI2_STAT under the null hypothesis.

The default is 99.

INITSEED (INPUT, OPTIONAL) logical(lgl) On entry, if INITSEED=true, a call to RANDOM_SEED_() without arguments is done in the subroutine, in order to initiates a non-repeatable reset of the seed used by the STATPACK random generator.

The default is INITSEED=false.

Further Details

This statistical randomization test relies on the assumptions that the different spectral ordinates have the same sampling distribution and are independent of each other. This means, in particular, that the spectral ordinates corresponding to the zero and Nyquist frequencies must be excluded from the PSVEC1 and PSVEC2 vectors before calling SPCTRM_DIFF2 and that the two estimated spectra have not been obtained by smoothing the periodograms in the frequency domain.

This randomization test could only be used to compare two periodograms or two spectral estimates computed as the the average of, say, r periodograms for each time series.

Finally, none of the spectral estimates must be zero.

For more details see:

(1) **Diggle, P.J., and Fisher, N.I., 1991:** Nonparametric comparison of cumulative periodograms, Applied Statistics, Vol. 40, No 3, pp. 423-434.

6.27.54 subroutine spctrm_diff2 (psmat1, psmat2, chi2_stat, prob, nrep, initseed)

Purpose

Subroutine SPCTRM_DIFF2 calculates approximate critical probability values (e.g. p-value) for testing the hypothesis of a common spectrum for two estimated multi-channel spectra (e.g. the arguments PSMAT1 and PSMAT2). These approximate critical probability values are derived from the following CHI2 statistics (e.g. the CHI2_STAT(:) argument) :

CHI2_STAT(:n) = (1/nf) [sum k=1 to nf] log(PSMAT1(:n,k) / PSMAT2(:n,k))**(2)

where n = size(PSMAT1,1) = size(PSMAT2,1) = size(CHI2_STAT) and nf = size(PSMAT2,2) = size(PSMAT2,2).

The distribution of CHI2_STAT under the null hypothesis of a common spectrum for the spectra of the two multi-channel series is approximated by calculating CHI2_STAT for some large number (e.g. the NREP argument) of random interchanges of periodogram ordinates at each frequency for the two estimated multi-channel spectra (e.g. the arguments PSMAT1(:,:) and PSMAT2(:,:)).

Arguments

PSMAT1 (INPUT) real(stnd), dimension(:,:) On entry, a real matrix containing the Power Spectral Density (PSD) estimates of the first multi-channel time series. Each row of the real matrix PSMAT1 contains the estimated spectrum of the corresponding "row" of the first multi-channel times series.

All elements in PSMAT1(:,:) must be greater than zero. size(PSMATN,2) must be greater or equal to 10.

PSMAT2 (INPUT) real(stnd), dimension(:,:) On entry, a real matrix containing the Power Spectral Density (PSD) estimates of the second multi-channel time series. Each row of the real matrix PSMAT2 contains the estimated spectrum of the corresponding "row" of the second multi-channel times series.

All elements in PSMAT2(:,:) must be greater than zero.

PSMAT2 must verify:

- size(PSMAT2,1) = size(PSMAT1,1) ,
- size(PSMAT2,2) = size(PSMAT2,2) .

CHI2_STAT (OUTPUT) real(stnd), dimension(:) On output, the CHI2 statistics computed from the multi-channel times series .

CHI2_STAT must verify: size(CHI2_STAT) = size(PSMAT1,1) .

PROB (OUTPUT) real(stnd), dimension(:) On exit, the aproximate critical probability values (e.g. p-values) computed under the hypothesis that the two "true" underlying multi-channel spectra are the same.

PROB is calculated as the probability of obtaining a value greater or equal to CHI2_STAT under the hypothesis of a common spectrum for the two multi-channel series.

PROB must verify: size(PROB) = size(PSMAT1,1) .

NREP (INPUT, OPTIONAL) integer(i4b) On entry, when argument NREP is present, NREP specifies the number of random interchanges of periodogram ordinates at each frequency in order to approximate the randomization distribution of CHI2_STAT under the null hypothesis.

The default is 99.

INITSEED (INPUT, OPTIONAL) logical(lgl) On entry, if INITSEED=true, a call to RANDOM_SEED_() without arguments is done in the subroutine, in order to initiate a non-repeatable reset of the seed used by the STATPACK random generator.

The default is INITSEED=false.

Further Details

This statistical randomization test relies on the assumptions that the different spectral ordinates have the same sampling distribution and are independent of each other. This means, in particular, that the spectral ordinates corresponding to the zero and Nyquist frequencies must be excluded from the PSMAT1 and PSMAT2 matrices before calling SPCTRM_DIFF2 and that the two estimated multi-channel spectra have not been obtained by smoothing the periodograms in the frequency domain.

This randomization test could only be used to compare two periodograms or two spectral estimates computed as the the average of, say, r periodograms for each time series.

Finally, none of the spectral estimates must be zero.

For more details see:

(1) **Diggle, P.J., and Fisher, N.I., 1991:** Nonparametric comparison of cumulative periodograms, Applied Statistics, Vol. 40, No 3, pp. 423-434.

6.27.55 subroutine power_spctrm (vec, psvec, freq, fftvec, edof, bandwidth, conlwr, conupr, initfft, normpsd, nsmooth, trend, win, taperp, probtest)

Purpose

Subroutine POWER_SPCTRM computes a Fast Fourier Transform (FFT) estimate of the power spectrum of a real time series, VEC. The real valued sequence VEC must be of even length.

The Power Spectral Density (PSD) estimates are returned in units which are the square of the data (if NORMPSD=false) or in spectral density units (if NORMPSD=true).

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the real time series for which the power spectrum must be estimated. If WIN/=2 or TREND=1, 2 or 3, VEC is used as workspace and is transformed.

Size(VEC) must be an even (positive) integer greater or equal to 4.

PSVEC (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the Power Spectral Density (PSD) estimates of VEC.

PSVEC must verify: size(PSVEC) = size(VEC)/2 + 1 .

FREQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the frequencies at which the spectral quantities are calculated in cycles per unit of time.

The spectral estimates are taken at frequencies (i-1)/size(VEC) for i=1,2, . . . , (size(VEC)/2 + 1).

FREQ must verify: size(FREQ) = size(VEC)/2 + 1 .

FFTVEC (OUTPUT, OPTIONAL) complex(stnd), dimension(:) On exit, a complex vector of length (size(VEC)/2)+1 containing the Fast Fourier Transform of the product of the (detrended, e.g. the TREND argument) real time series VEC with the choosen window function (e.g. The WIN argument).

FFTVEC must verify: size(FFTVEC) = size(VEC)/2 + 1 .

EDOF (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the equivalent number of degrees of freedom of the power spectrum estimates.

EDOF must verify: size(EDOF) = size(VEC)/2 + 1 .

BANDWIDTH (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the bandwidth of the power spectrum estimates.

BANDWIDTH must verify: size(BANDWIDTH) = size(VEC)/2 + 1 .

CONLWR (OUTPUT, OPTIONAL) real(stnd), dimension(:)

CONUPR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On output, these arguments specify the lower and upper (1-PROBTEST) * 100% confidence limit factors, respectively. Multiply the PSD

estimates (e.g. the PSVEC(:) argument) by these constants to get the lower and upper limits of a (1-PROBTEST) * 100% confidence interval for the PSD estimates.

CONLWR must verify: size(CONLWR) = size(VEC)/2 + 1 .

CONUPR must verify: size(CONUPR) = size(VEC)/2 + 1 .

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITFFT = false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine POWER_SPCTRM in order to sets up constants and functions for use by subroutine FFT which is called inside subroutine POWER_SPCTRM. This call to INITFFT must have the following form:

  call init_fft(size(VEC)/2)

- INITFFT = true, the call to INIT_FFT is done inside subroutine POWER_SPCTRM and a call to END_FFT is also done before leaving subroutine POWER_SPCTRM.

The default is INITFFT=true .

NORMPSD (INPUT, OPTIONAL) logical(lgl) On entry, if:

- NORMPSD = true, the PSD estimates are normalized in such a way that the total area under the power spectrum is equal to the variance of the time series VEC.
- NORMPSD = false, the sum of the PSD estimates (e.g. sum(PSVEC(2:)) is equal to the variance of the time series.

The default is NORMPSD=true .

NSMOOTH (INPUT, OPTIONAL) integer(i4b) if NSMOOTH is used, the PSD estimates are computed by smoothing the periodogram with Daniell weights (e.g. a simple moving average).

On entry, NSMOOTH gives the length of the Daniell filter to be applied.

Setting NSMOOTH=0 on entry is equivalent to omit the optional argument NSMOOTH. Otherwise, NSMOOTH must be odd, greater than 2 and less or equal to size(VEC)/2+1 .

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+1 The mean of the time series is removed before computing the spectrum
- TREND=+2 The drift from the time series is removed before computing the spectrum by using the formula: drift = (VEC(size(VEC)) - VEC(1))/(size(VEC) - 1)
- TREND=+3 The least-squares line from the time series is removed before computing the spectrum.

For other values of TREND nothing is done before estimating the power spectrum.

The default is TREND=1, e.g. the mean of the time series is removed before the computations.

WIN (INPUT, OPTIONAL) integer(i4b) On entry, this argument specify the data window used in the computations of the power spectrum. If:

- WIN=+1 The Bartlett window is used
- WIN=+2 The square window is used
- WIN=+3 The Welch window is used
- WIN=+4 The Hann window is used
- WIN=+5 The Hamming window is used
- WIN=+6 A split-cosine-bell window is used

The default is WIN=3, e.g. the Welch window is used.

TAPERP (INPUT, OPTIONAL) real(stnd) The total percentage of the data to be tapered if WIN=6. TAPERP must be greater than zero and less or equal to one, otherwise the default value is used.

The default is 0.2 .

PROBTEST (INPUT, OPTIONAL) real(stnd) On entry, a probability. PROBTEST is the critical probability which is used to determine the lower and upper confidence limit factors (e.g. the optional arguments CONLWR and CONUPR).

PROBTEST must verify: 0. < P < 1.

The default is 0.05 .

Further Details

After removing the mean or the trend from the time series (e.g. TREND=1,2,3), the selected data window (e.g. WIN=1,2,3,4,5,6) is applied to the time series and the PSD estimates are computed by the FFT of this transformed time series. Optionally, theses PSD estimates may then be smoothed in the frequency domain by a Daniell filter (e.g. if NSMOOTH is used).

For definitions, more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York.

(2) **Welch, P.D., 1967:** The use of Fast Fourier Transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. IEEE trans. on audio and electroacoustics, Vol. Au-15, 2, 70-73.

(3) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford.

6.27.56 subroutine power_spctrm (mat, psmat, freq, fftmat, edof, bandwidth, conlwr, conupr, initfft, normpsd, nsmooth, trend, win, taperp, probtest)

Purpose

Subroutine POWER_SPCTRM computes a Fast Fourier Transform (FFT) estimate of the power spectra of the rows of the real matrix, MAT. size(MAT,2) must be of even length.

The Power Spectral Density (PSD) estimates are returned in units which are the square of the data (if NORMPSD=false) or in spectral density units (if NORMPSD=true).

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real time series for which power spectra must be estimated. Each row of MAT is a real time series.

If WIN/=2 or TREND=1, 2 or 3, MAT is used as workspace and is transformed.

Size(MAT,2) must be an even (positive) integer greater or equal to 4.

PSMAT (OUTPUT) real(stnd), dimension(:,:) On exit, a real matrix containing the Power Spectral Density (PSD) estimates for each row of the real matrix MAT.

The shape of PSMAT must verify:

- size(PSMAT,1) = size(MAT,1) ;
- size(PSMAT,2) = size(MAT,2)/2 + 1 .

FREQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length (size(MAT,2)/2)+1 containing the frequencies at which the spectral quantities are calculated in cycles per unit of time.

The spectral estimates are taken at frequencies (i-1)/size(VEC) for i=1,2, ... , (size(MAT,2)/2 + 1).

FREQ must verify: size(FREQ) = size(MAT,2)/2 + 1 .

FFTMAT (OUTPUT, OPTIONAL) complex(stnd), dimension(:,:) On exit, a complex matrix containing the Fast Fourier Transform of the product of the (detrended, e.g. the TREND argument) real time series in each row of MAT with the choosen window function (e.g. The WIN argument).

The shape of FFTMAT must verify:

- size(FFTMAT,1) = size(MAT,1) ;
- size(FFTMAT,2) = size(MAT,2)/2 + 1 .

EDOF (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the equivalent number of degrees of freedom of the power spectrum estimates.

EDOF must verify: size(EDOF) = size(MAT,2)/2 + 1 .

BANDWIDTH (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the bandwidth of the power spectrum estimates.

BANDWIDTH must verify: size(BANDWIDTH) = size(MAT,2)/2 + 1 .

CONLWR (OUTPUT, OPTIONAL) real(stnd), dimension(:)

CONUPR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On output, these arguments specify the lower and upper (1-PROBTEST) * 100% confidence limit factors, respectively. Multiply the PSD estimates (e.g. the PSMAT(:,:) argument) by these constants to get the lower and upper limits of a (1-PROBTEST) * 100% confidence interval for the PSD estimates.

CONLWR must verify: size(CONLWR) = size(MAT,2)/2 + 1 .

CONUPR must verify: size(CONUPR) = size(MAT,2)/2 + 1 .

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITFFT = false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine POWER_SPCTRM in order to sets up constants and functions for use by subroutine FFT which is called inside subroutine POWER_SPCTRM. This call to INITFFT must have the following form:

  call init_fft((/ size(MAT,1), size(MAT,2)/2 /), dim=2_i4b)

- INITFFT = true, the call to INIT_FFT is done inside subroutine POWER_SPCTRM and a call to END_FFT is also done before leaving subroutine POWER_SPCTRM

The default is INITFFT=true .

NORMPSD (INPUT, OPTIONAL) logical(lgl) On entry, if:

- NORMPSD is set to true, the PSD estimates are normalized in such a way that the total area under the power spectrum is equal to the variance of the time series MAT.
- NORMPSD = false, the sum of the PSD estimates for each row of MAT (e.g. sum(PSMAT(:,2:), dim=2) is equal to the variance of the corresponding time series.

The default is NORMPSD=true .

NSMOOTH (INPUT, OPTIONAL) integer(i4b) if NSMOOTH is used, the PSD estimates are computed by smoothing the periodogram with Daniell weights (e.g. a simple moving average).

On entry, NSMOOTH gives the length of the Daniell filter to be applied.

Setting NSMOOTH=0 on entry is equivalent to omit the optional argument NSMOOTH. Otherwise, NSMOOTH must be odd, greater than 2 and less or equal to size(MAT,2)/2+1 .

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+1 The means of the time series are removed before computing the spectra
- TREND=+2 The drifts from the time series are removed before computing the spectra by using the formula: drift(i) = (MAT(i,size(MAT,2)) - MAT(i,1))/(size(MAT,2) - 1)
- TREND=+3 The least-squares lines from the time series are removed before computing the spectra.

For other values of TREND nothing is done before estimating the power spectra.

The default is TREND=1, e.g. the means of the time series are removed before the computations.

WIN (INPUT, OPTIONAL) integer(i4b) On entry, this argument specify the data window used in the computations of the power spectrum. If:

- WIN=+1 The Bartlett window is used
- WIN=+2 The square window is used
- WIN=+3 The Welch window is used
- WIN=+4 The Hann window is used
- WIN=+5 The Hamming window is used
- WIN=+6 A split-cosine-bell window is used WIN=+1 The Bartlett window is used

The default is WIN=3, e.g. the Welch window is used.

TAPERP (INPUT, OPTIONAL) real(stnd) The total percentage of the data to be tapered if WIN=6. TAPERP must be greater than zero and less or equal to one, otherwise the default value is used.

The default is 0.2 .

PROBTEST (INPUT, OPTIONAL) real(stnd) On entry, a probability. PROBTEST is the critical probability which is used to determine the lower and upper confidence limit factors (e.g. the optional arguments CONLWR and CONUPR).

PROBTEST must verify: 0. < P < 1.

The default is 0.05 .

Further Details

After removing the mean or the trend from the time series (e.g. TREND=1,2,3), the selected data window (e.g. WIN=1,2,3,4,5,6) is applied to the time series and the PSD estimates are computed by the FFT of these transformed time series. Optionally, theses PSD estimates may then be smoothed in the frequency domain by modified Daniell filters (e.g. if SMOOTH_PARAM is used).

For definitions, more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York.

(2) **Welch, P.D., 1967:** The use of Fast Fourier Transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. IEEE trans. on audio and electroacoustics, Vol. Au-15, 2, 70-73.

(3) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford.

6.27.57 subroutine cross_spctrm (vec, vec2, psvec, psvec2, phase, coher, freq, edof, bandwidth, conlwr, conupr, testcoher, ampli, co_spect, quad_spect, prob_coher, initfft, normpsd, nsmooth, trend, win, taperp, probtest)

Purpose

Subroutine CROSS_SPCTRM computes Fast Fourier Transform (FFT) estimates of the power and cross spectra of two real time series, VEC and VEC2. The real valued sequences VEC and VEC2 must be of even length.

The Power Spectral Density (PSD) and Cross Spectral Density (CSD) estimates are returned in units which are the square of the data (if NORMPSD=false) or in spectral density units (if NORMPSD=true).

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the first real time series for which the power and cross spectra must be estimated. If WIN/=2 or TREND=1, 2 or 3, VEC is used as workspace and is transformed.

Size(VEC) must be an even (positive) integer greater or equal to 4.

VEC2 (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the second real time series for which the power and cross spectra must be estimated. If WIN/=2 or TREND=1, 2 or 3, VEC2 is used as workspace and is transformed.

VEC2 must verify: size(VEC2) = size(VEC).

PSVEC (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the Power Spectral Density (PSD) estimates of VEC.

PSVEC must verify: size(PSVEC) = size(VEC)/2 + 1 .

PSVEC2 (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length (size(VEC2)/2)+1 containing the Power Spectral Density (PSD) estimates of VEC2.

PSVEC2 must verify: size(PSVEC2) = size(VEC)/2 + 1 .

PHASE (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the phase of the cross spectrum, given in fractions of a circle (e.g. on the closed interval (0,1)).

PHASE must verify: size(PHASE) = size(VEC)/2 + 1 .

COHER (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the squared coherency estimates for all frequencies.

COHER must verify: size(COHER) = size(VEC)/2 + 1 .

FREQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the frequencies at which the spectral quantities are calculated in cycles per unit of time.

The spectral estimates are taken at frequencies (i-1)/size(VEC) for i=1,2, . . . , (size(VEC)/2 + 1).

FREQ must verify: size(FREQ) = size(VEC)/2 + 1 .

EDOF (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the equivalent number of degrees of freedom of the power spectrum estimates.

EDOF must verify: size(EDOF) = size(VEC)/2 + 1 .

BANDWIDTH (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the bandwidth of the power spectrum estimates.

BANDWIDTH must verify: size(BANDWIDTH) = size(VEC)/2 + 1 .

CONLWR (OUTPUT, OPTIONAL) real(stnd), dimension(:)

CONUPR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On output, these arguments specify the lower and upper (1-PROBTEST) * 100% confidence limit factors, respectively. Multiply the PSD estimates (e.g. the PSVEC(:) and PSVEC2(:) arguments) by these constants to get the lower and upper limits of a (1-PROBTEST) * 100% confidence interval for the PSD estimates.

CONLWR must verify: size(CONLWR) = size(VEC)/2 + 1 .

CONUPR must verify: size(CONUPR) = size(VEC)/2 + 1 .

TESTCOHER (OUTPUT, OPTIONAL) real(stnd), dimension(:) On output, this argument specifies the critical value for testing the null hypothesis that the squared coherency is zero at the PROBTEST * 100% significance level (e.g. elements of COHER(:) less than TESTCOHER(:) should be regarded as not significantly different from zero at the PROBTEST * 100% significance level).

TESTCOHER must verify: size(TESTCOHER) = size(VEC)/2 + 1 .

AMPLI (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the cross-amplitude spectrum.

AMPLI must verify: size(AMPLI) = (size(VEC)/2) + 1 .

CO_SPECT (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the co-spectrum (e.g. the real part of cross-spectrum).

CO_SPECT must verify: size(CO_SPECT) = (size(VEC)/2) + 1 .

QUAD_SPECT (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the quadrature spectrum (e.g. the imaginary part of cross-spectrum with a minus sign).

QUAD_SPECT must verify: size(QUAD_SPECT) = (size(VEC)/2) + 1 .

PROB_COHER (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the probabilities that the computed sample squared coherencies came from an ergodic stationary bivariate process with (corresponding) squared coherencies equal to zero.

PROB_COHER must verify: size(PROB_COHER) = (size(VEC)/2) + 1 .

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITFFT = false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine CROSS_SPCTRM in order to sets up constants and functions for use by subroutine FFT which is called inside subroutine CROSS_SPCTRM. This call to INITFFT must have the following form:

  call init_fft(size(VEC)/2)

- INITFFT is set to true, the call to INIT_FFT is done inside subroutine CROSS_SPCTRM and a call to END_FFT is also done before leaving subroutine CROSS_SPCTRM.

The default is INITFFT=true .

NORMPSD (INPUT, OPTIONAL) logical(lgl) On entry, if:

- NORMPSD = true, the power and cross spectra estimates are normalized in such a way that the total area under the power spectrum is equal to the variance of the time series VEC and VEC2.
- NORMPSD = false, the sum of the PSD estimates (e.g. sum(PSVEC(2:)) and sum(PSVEC2(2:))) is equal to the variance of the corresponding time series.

The default is NORMPSD=true .

NSMOOTH (INPUT, OPTIONAL) integer(i4b) If NSMOOTH is used, the PSD and CSD estimates are computed by smoothing the periodogram with Daniell weights (e.g. a simple moving average).

On entry, NSMOOTH gives the length of the Daniell filter to be applied.

Setting NSMOOTH=0 on entry is equivalent to omit the optional argument NSMOOTH. Otherwise, NSMOOTH must be odd, greater than 2 and less or equal to size(VEC)/2+1 .

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+1 The mean of the two time series is removed before computing the power and cross spectra.
- TREND=+2 The drift from the two time series is removed before computing the power and cross spectra.
- TREND=+3 The least-squares line from the two time series is removed before computing the power and cross spectra.

For other values of TREND nothing is done before estimating the power and cross spectra.

The default is TREND=1, e.g. the means of the time series are removed before the computations.

WIN (INPUT, OPTIONAL) integer(i4b) On entry, this argument specify the data window used in the computations of the power and cross spectra. If:

- WIN=+1 The Bartlett window is used
- WIN=+2 The square window is used
- WIN=+3 The Welch window is used
- WIN=+4 The Hann window is used
- WIN=+5 The Hamming window is used
- WIN=+6 A split-cosine-bell window is used

The default is WIN=3, e.g. the Welch window is used.

TAPERP (INPUT, OPTIONAL) real(stnd) The total percentage of the data to be tapered if WIN=6. TAPERP must be greater than zero and less or equal to one, otherwise the default value is used.

The default is 0.2 .

PROBTEST (INPUT, OPTIONAL) real(stnd) On entry, a probability. PROBTEST is the critical probability which is used to determine the lower and upper confidence limit factors (e.g. the optional arguments CONLWR and CONUPR) and the critical value for testing the null hypothesis that the squared coherency is zero (e.g. the TESTCOHER optional argument).

PROBTEST must verify: 0. < P < 1.

The default is 0.05 .

Further Details

After removing the mean or the trend from the time series (e.g. TREND=1,2,3), the selected data window (e.g. WIN=1,2,3,4,5,6) is applied to the time series and the PSD and CSD estimates are computed by the FFT of these transformed time series. Optionally, theses PSD and CSD estimates may then be smoothed in the frequency domain by modified Daniell filters (e.g. if argument NSMOOTH is used).

For definitions, more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York.

(2) **Welch, P.D., 1967:** The use of Fast Fourier Transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. IEEE trans. on audio and electroacoustics, Vol. Au-15, 2, 70-73.

(3) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford.

6.27.58 subroutine cross_spctrm (vec, mat, psvec, psmat, phase, coher, freq, edof, bandwidth, conlwr, conupr, testcoher, ampli, co_spect, quad_spect, prob_coher, initfft, normpsd, nsmooth, trend, win, taperp, probtest)

Purpose

Subroutine CROSS_SPCTRM computes Fast Fourier Transform (FFT) estimates of the power and cross spectra of the real time series, VEC, and the multi-channel real time series MAT.

The Power Spectral Density (PSD) and Cross Spectral Density (CSD) estimates are returned in units which are the square of the data (if NORMPSD=false) or in spectral density units (if NORMPSD=true).

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the real time series for which the power and cross spectra must be estimated. If WIN/=2 or TREND=1, 2 or 3, VEC is used as workspace and is transformed.

Size(VEC) must be an even (positive) integer greater or equal to 4.

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the multi-channel real time series for which the power and cross spectra must be estimated. Each row of MAT is a real time series. If WIN/=2 or TREND=1, 2 or 3, MAR is used as workspace and is transformed.

The shape of MAT must verify: size(MAT,2) = size(VEC).

PSVEC (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the Power Spectral Density (PSD) estimates of VEC.

PSVEC must verify: size(PSVEC) = size(VEC)/2 + 1 .

PSMAT (OUTPUT) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and (size(VEC)/2)+1 columns containing the Power Spectral Density (PSD) estimates of each row of MAT.

The shape of PSMAT must verify:

- size(PSMAT,1) = size(MAT,1) ;
- size(PSMAT,2) = size(VEC)/2 + 1 .

PHASE (OUTPUT) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and (size(VEC)/2)+1 columns containing the phase of the cross spectrum, given in fractions of a circle (e.g. on the closed interval (0,1)).

The shape of PHASE must verify:

- size(PHASE,1) = size(MAT,1) ;
- size(PHASE,2) = size(VEC)/2 + 1 .

COHER (OUTPUT) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and (size(VEC)/2)+1 columns containing the squared coherency estimates for all frequencies.

The shape of COHER must verify:

- size(COHER,1) = size(MAT,1) ;
- size(COHER,2) = size(VEC)/2 + 1 .

FREQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the frequencies at which the spectral quantities are calculated in cycles per unit of time.

The spectral estimates are taken at frequencies (i-1)/size(VEC) for i=1,2, . . . , (size(VEC)/2 + 1).

FREQ must verify: size(FREQ) = size(VEC)/2 + 1 .

EDOF (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the equivalent number of degrees of freedom of the power spectrum estimates.

EDOF must verify: size(EDOF) = size(VEC)/2 + 1 .

BANDWIDTH (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the bandwidth of the power spectrum estimates.

BANDWIDTH must verify: size(BANDWIDTH) = size(VEC)/2 + 1 .

CONLWR (OUTPUT, OPTIONAL) real(stnd), dimension(:)

CONUPR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On output, these arguments specify the lower and upper (1-PROBTEST) * 100% confidence limit factors, respectively. Multiply the PSD estimates (e.g. the PSVEC(:) and PSMAT(:,:) arguments) by these constants to get the lower and upper limits of a (1-PROBTEST) * 100% confidence interval for the PSD estimates.

CONLWR must verify: size(CONLWR) = size(VEC)/2 + 1 .

CONUPR must verify: size(CONUPR) = size(VEC)/2 + 1 .

TESTCOHER (OUTPUT, OPTIONAL) real(stnd), dimension(:) On output, this argument specifies the critical value for testing the null hypothesis that the squared coherency is zero at the PROBTEST * 100% significance level (e.g. elements of COHER(:,:) less than TESTCOHER(:) should be regarded as not significantly different from zero at the PROBTEST * 100% significance level).

TESTCOHER must verify: size(TESTCOHER) = size(VEC)/2 + 1 .

AMPLI (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and (size(VEC)/2)+1 columns containing the cross-amplitude spectra.

The shape of AMPLI must verify:

- size(AMPLI,1) = size(MAT,1) ;
- size(AMPLI,2) = (size(VEC)/2) + 1 .

CO_SPECT (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and (size(VEC)/2)+1 columns containing the co-spectra (e.g. the real part of cross-spectra).

The shape of CO_SPECT must verify:

- size(CO_SPECT,1) = size(MAT,1) ;
- size(CO_SPECT,2) = (size(VEC)/2) + 1 .

QUAD_SPECT (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and (size(VEC)/2)+1 columns containing the quadrature spectrum (e.g. the imaginary part of cross-spectrum with a minus sign).

The shape of QUAD_SPECT must verify:

- size(QUAD_SPECT,1) = size(MAT,1) ;
- size(QUAD_SPECT,2) = (size(VEC)/2) + 1 .

PROB_COHER (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and (size(VEC)/2)+1 columns containing the probabilities that the computed sample squared coherencies came from an ergodic stationary bivariate process with (corresponding) squared coherencies equal to zero.

The shape of PROB_COHER must verify:

- size(PROB_COHER,1) = size(MAT,1) ;
- size(PROB_COHER,2) = (size(VEC)/2) + 1 .

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITFFT = false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine CROSS_SPCTRM in order to sets up constants and functions for use by subroutine FFT which is called inside subroutine CROSS_SPCTRM. This call to INITFFT must have the following form:

  call init_fft((/ size(MAT,1), size(MAT,2)/2 /), dim=2_i4b)

- INITFFT = true, the call to INIT_FFT is done inside subroutine CROSS_SPCTRM and a call to END_FFT is also done before leaving subroutine CROSS_SPCTRM.

The default is INITFFT=true .

NORMPSD (INPUT, OPTIONAL) logical(lgl) On entry, if:

- NORMPSD = true, the power and cross spectra estimates are normalized in such a way that the total area under the power spectra is equal to the variance of the time series contained in VEC and in each row of MAT.
- NORMPSD = false, the sum of the PSD estimates (e.g. sum(PSVEC(2:)) and sum(PSMAT(:,2:),dim=2)) is equal to the variance of the corresponding time series.

The default is NORMPSD=true .

NSMOOTH (INPUT, OPTIONAL) integer(i4b) If NSMOOTH is used, the PSD and CSD estimates are computed by smoothing the periodogram with Daniell weights (e.g. a simple moving average).

On entry, NSMOOTH gives the length of the Daniell filter to be applied.

Setting NSMOOTH=0 on entry is equivalent to omit the optional argument NSMOOTH. Otherwise, NSMOOTH must be odd, greater than 2 and less or equal to size(VEC)/2+1 .

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+1 The means of the time series are removed before computing the power and cross spectra
- TREND=+2 The drifts from time series are removed before computing the power and cross spectra
- TREND=+3 The least-squares lines from time series are removed before computing the power and cross spectra.

For other values of TREND nothing is done before estimating the power and cross spectra.

The default is TREND=1, e.g. the means of the time series are removed before the computations.

WIN (INPUT, OPTIONAL) integer(i4b) On entry, this argument specify the data window used in the computations of the power and cross spectra. If:

- WIN=+1 The Bartlett window is used
- WIN=+2 The square window is used
- WIN=+3 The Welch window is used
- WIN=+4 The Hann window is used
- WIN=+5 The Hamming window is used
- WIN=+6 A split-cosine-bell window is used

The default is WIN=3, e.g. the Welch window is used.

TAPERP (INPUT, OPTIONAL) real(stnd) The total percentage of the data to be tapered if WIN=6. TAPERP must be greater than zero and less or equal to one, otherwise the default value is used.

The default is 0.2 .

PROBTEST (INPUT, OPTIONAL) real(stnd) On entry, a probability. PROBTEST is the critical probability which is used to determine the lower and upper confidence limit factors (e.g. the optional arguments CONLWR and CONUPR) and the critical value for testing the null hypothesis that the squared coherency is zero (e.g. the TESTCOHER optional argument).

PROBTEST must verify: 0. < P < 1.

The default is 0.05 .

Further Details

After removing the mean or the trend from the time series (e.g. TREND=1,2,3), the selected data window (e.g. WIN=1,2,3,4,5,6) is applied to the time series and the PSD and CSD estimates are computed by the FFT of these transformed time series. Optionally, theses PSD and CSD estimates may then be smoothed in the frequency domain by modified Daniell filters (e.g. if argument NSMOOTH is used).

For definitions, more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York.

(2) **Welch, P.D., 1967:** The use of Fast Fourier Transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. IEEE trans. on audio and electroacoustics, Vol. Au-15, 2, 70-73.

(3) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford.

6.27.59 subroutine power_spctrm2 (vec, l, psvec, freq, edof, bandwidth, conlwr, conupr, initfft, overlap, normpsd, nsmooth, trend, trend2, win, taperp, l0, probtest)

Purpose

Subroutine POWER_SPCTRM2 computes a Fast Fourier Transform (FFT) estimate of the power spectrum of a real time series.

The Power Spectral Density (PSD) estimates are returned in units which are the square of the data (if NORMPSD=false) or in spectral density units (if NORMPSD=true).

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the real time series for which the power spectrum must be estimated. If TREND=1, 2 or 3, VEC is used as workspace and is transformed.

Size(VEC) must be greater or equal to 4.

L (INPUT) integer(i4b) On entry, an integer used to segment the time series. L is the length of the segments. L must be a positive even integer, less or equal to size(VEC), but greater or equal to 4.

Spectral computations are at (L/2)+1 frequencies if the optional argument L0 is absent and are at ((L+L0)/2)+1 frequencies if L0 is present (L0 is the number of zeros added to each segment).

Suggested values for L+L0 are 16, 32, 64 or 128 (e.g. an integer power of two, in order to speed the computations).

PSVEC (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the Power Spectral Density (PSD) estimates of VEC.

PSVEC must verify: size(PSVEC) = ((L+L0)/2) + 1 .

FREQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the frequencies at which the spectral quantities are calculated in cycles per unit of time.

The spectral estimates are taken at frequencies (i-1)/(L+L0) for i=1,2, . . . , ((L+L0)/2 + 1).

FREQ must verify: size(FREQ) = (L+L0)/2 + 1 .

EDOF (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the equivalent number of degrees of freedom of the power spectrum estimates.

EDOF must verify: size(EDOF) = ((L+L0)/2) + 1 .

BANDWIDTH (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the bandwidth of the power spectrum estimates.

BANDWIDTH must verify: size(BANDWIDTH) = ((L+L0)/2) + 1 .

CONLWR (OUTPUT, OPTIONAL) real(stnd), dimension(:)

CONUPR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On output, these arguments specify the lower and upper (1-PROBTEST) * 100% confidence limit factors, respectively. Multiply the PSD estimates (e.g. the PSVEC(:) argument) by these constants to get the lower and upper limits of a (1-PROBTEST) * 100% confidence interval for the PSD estimates.

CONLWR must verify: size(CONLWR) = ((L+L0)/2) + 1 .

CONUPR must verify: size(CONUPR) = ((L+L0)/2) + 1 .

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITFFT = false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine POWER_SPCTRM2 in order to sets up constants and functions for use by subroutine FFT which is called inside subroutine POWER_SPCTRM2. This call to INITFFT must have the following form:

  call init_fft((L+L0)/2)

- INITFFT = true, the call to INIT_FFT is done inside subroutine POWER_SPCTRM2 and a call to END_FFT is also done before leaving subroutine POWER_SPCTRM2.

The default is INITFFT=true .

OVERLAP (INPUT, OPTIONAL) logical(lgl) If:

- OVERLAP = false, the subroutine segments the data without any overlapping.
- OVERLAP=true, the subroutine overlaps the segments by one half of their length (which is equal to L).

In both cases, zeros are eventually added to each segment (if argument L0 is present) and each segment will be FFT'd, and the resulting periodograms will averaged together to obtain a Power Spectrum Density estimate at the ((L+L0)/2)+1 frequencies.

The default is OVERLAP=false .

NORMPSD (INPUT, OPTIONAL) logical(lgl) On entry, if:

- NORMPSD = true, the PSD estimates are normalized in such a way that the total area under the power spectrum is equal to the variance of the time series VEC.
- NORMPSD is set to false, the sum of the PSD estimates (e.g. sum(PSVEC(2:)) is equal to the variance of the time series.

The default is NORMPSD=true .

NSMOOTH (INPUT, OPTIONAL) integer(i4b) If NSMOOTH is used, the PSD estimates are computed by smoothing the periodogram with Daniell weights (e.g. a simple moving average).

On entry, NSMOOTH gives the length of the Daniell filter to be applied.

Setting NSMOOTH=0 on entry is equivalent to omit the optional argument NSMOOTH. Otherwise, NSMOOTH must be odd, greater than 2 and less or equal to (L+L0)/2+1 .

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+1 The mean of the time series is removed before computing the spectrum
- TREND=+2 The drift from the time series is removed before computing the spectrum by using the formula: drift = (VEC(size(VEC)) - VEC(1))/(size(VEC) - 1)
- TREND=+3 The least-squares line from the time series is removed before computing the spectrum.

For other values of TREND nothing is done before estimating the power spectrum.

The default is TREND=1, e.g. the mean of the time series is removed before the computations.

TREND2 (INPUT, OPTIONAL) integer(i4b) If:

- TREND2=+1 The mean of the time segment is removed before computing the spectrum on this segment.
- TREND2=+2 The drift from the time segment is removed before computing the spectrum on this segment.

- TREND2=+3 The least-squares line from the time segment is removed before computing the spectrum on this segment.

For other values of TREND2 nothing is done before estimating the power spectrum on each segment.

The default is TREND2=0, e.g. nothing is done before estimating the power spectrum on each segment.

WIN (INPUT, OPTIONAL) integer(i4b) On entry, this argument specify the data window used in the computations of the power spectrum. If:

- WIN=+1 The Bartlett window is used
- WIN=+2 The square window is used
- WIN=+3 The Welch window is used
- WIN=+4 The Hann window is used
- WIN=+5 The Hamming window is used
- WIN=+6 A split-cosine-bell window is used

The default is WIN=3, e.g. the Welch window is used.

TAPERP (INPUT, OPTIONAL) real(stnd) The total percentage of the data to be tapered if WIN=6. TAPERP must be greater than zero and less or equal to one, otherwise the default value is used.

The default is 0.2 .

L0 (INPUT, OPTIONAL) integer(i4b) The number of zeros added to each time segment in order to obtain more finely spaced spectral estimates. L+L0 must be a positive even integer.

The default is L0=0, e.g. no zeros are added to each time segment.

PROBTEST (INPUT, OPTIONAL) real(stnd) On entry, a probability. PROBTEST is the critical probability which is used to determine the lower and upper confidence limit factors (e.g. the optional arguments CONLWR and CONUPR).

PROBTEST must verify: 0. < P < 1.

The default is 0.05 .

Further Details

After removing the mean or the trend from the time series (e.g. TREND=1,2,3), the series is padded with zero on the right such that the length of the resulting time series is evenly divisible by L (a positive even integer). The length, N, of this resulting time series is the first integer greater than or equal to size(VEC) which is evenly divisible by L. If size(VEC) is not evenly divisible by L, N is equal to size(VEC)+L-mod(size(VEC),L).

Optionally, the mean or the trend may also be removed from each time segment (e.g. TREND2=1,2,3). Optionally, zeros may be added to each time segment (e.g. the optional arguemnt L0) if more finely spaced spectral esimates are desired.

The stability of the PSD estimates depends on the averaging process. That is, the greater the number of segments (N/L if OVERLAP=false and (2N/L)-1 if OVERLAP=true), the more stable the resulting PSD estimates.

Optionally, theses PSD estimates may then be smoothed again in the frequency domain by a Daniell filter (e.g. if argument NSMOOTH is used).

For definitions, more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York.

(2) **Welch, P.D., 1967:** The use of Fast Fourier Transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. IEEE trans. on audio and electroacoustics, Vol. Au-15, 2, 70-73.

(3) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford.

6.27.60 subroutine power_spctrm2 (mat, l, psmat, freq, edof, bandwidth, conlwr, conupr, initfft, overlap, normpsd, nsmooth, trend, trend2, win, taperp, l0, probtest)

Purpose

Subroutine POWER_SPCTRM2 computes Fast Fourier Transform (FFT) estimates of the power spectra of the multi-channel real time series MAT (e.g. each row of MAT contains a time series).

The Power Spectral Density (PSD) estimates are returned in units which are the square of the data (if NORMPSD=false) or in spectral density units (if NORMPSD=true).

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the multi-channel real time series for which the power spectra must be estimated. Each row of MAT is a real time series. If TREND=1, 2 or 3, MAT is used as workspace and is transformed.

Size(MAT,2) must be greater or equal to 4.

L (INPUT) integer(i4b) On entry, an integer used to segment the time series. L is the length of the segments. L must be a positive even integer less or equal to size(MAT,2), but greater or equal to 4.

Spectral computations are at (L/2)+1 frequencies if the optional argument L0 is absent and are at ((L+L0)/2)+1 frequencies if L0 is present (L0 is the number of zeros added to each segment).

Suggested values for L+L0 are 16, 32, 64 or 128 (e.g. an integer power of two, in order to speed the computations).

PSMAT (OUTPUT) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and ((L+L0)/2) + 1 columns containing the Power Spectral Density (PSD) estimates of each row of MAT.

The shape of PSMAT must verify:

- size(PSMAT,1) = size(MAT,1) ;
- size(PSMAT,2) = ((L+L0)/2) + 1 .

FREQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the frequencies at which the spectral quantities are calculated in cycles per unit of time.

The spectral estimates are taken at frequencies (i-1)/(L+L0) for i=1,2, . . . , ((L+L0)/2 + 1).

FREQ must verify: size(FREQ) = (L+L0)/2 + 1 .

EDOF (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the equivalent number of degrees of freedom of the power spectrum estimates.

EDOF must verify: size(EDOF) = ((L+L0)/2) + 1 .

BANDWIDTH (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the bandwidth of the power spectrum estimates.

BANDWIDTH must verify: size(BANDWIDTH) = ((L+L0)/2) + 1 .

CONLWR (OUTPUT, OPTIONAL) real(stnd), dimension(:)

CONUPR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On output, these arguments specify the lower and upper (1-PROBTEST) * 100% confidence limit factors, respectively. Multiply the PSD estimates (e.g. the PSMAT(:,:) argument) by these constants to get the lower and upper limits of a (1-PROBTEST) * 100% confidence interval for the PSD estimates.

CONLWR must verify: size(CONLWR) = ((L+L0)/2) + 1 .

CONUPR must verify: size(CONUPR) = ((L+L0)/2) + 1 .

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITFFT = false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine POWER_SPCTRM2 in order to sets up constants and functions for use by subroutine FFT which is called inside subroutine POWER_SPCTRM2. This call to INITFFT must have the following form:

  call init_fft((/ size(MAT,1), (L+L0)/2 /), dim=2_i4b)

- INITFFT = true, the call to INIT_FFT is done inside subroutine POWER_SPCTRM2 and a call to END_FFT is also done before leaving subroutine POWER_SPCTRM2.

The default is INITFFT=true .

OVERLAP (INPUT, OPTIONAL) logical(lgl) If:

- OVERLAP = false, the subroutine segments the data without any overlapping.
- OVERLAP=true, the subroutine overlaps the segments by one half of their length (which is equal to L).

In both cases, zeros are eventually added to each segment (if argument L0 is present) and each segment will be FFT'd, and the resulting periodograms will averaged together to obtain a Power Spectrum Density estimate at the ((L+L0)/2)+1 frequencies.

The default is OVERLAP=false .

NORMPSD (INPUT, OPTIONAL) logical(lgl) On entry, if:

- NORMPSD = true, the PSD estimates are normalized in such a way that the total area under the power spectrum is equal to the variance of the time series VEC.
- NORMPSD is set to false, the sum of the PSD estimates (e.g. sum(PSMAT(:,2:),dim=2)) is equal to the variance of the corresponding time series.

The default is NORMPSD=true .

NSMOOTH (INPUT, OPTIONAL) integer(i4b) If NSMOOTH is used, the PSD estimates are computed by smoothing the periodogram with Daniell weights (e.g. a simple moving average).

On entry, NSMOOTH gives the length of the Daniell filter to be applied.

Setting NSMOOTH=0 on entry is equivalent to omit the optional argument NSMOOTH. Otherwise, NSMOOTH must be odd, greater than 2 and less or equal to (L+L0)/2+1 .

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+1 The means of the time series are removed before computing the spectra
- TREND=+2 The drifts from time series are removed before computing the spectra

- TREND=+3 The least-squares lines from time series are removed before computing the spectra.

For other values of TREND nothing is done before estimating the power and cross spectra. The default is TREND=1, e.g. the means of the time series are removed before the computations.

TREND2 (INPUT, OPTIONAL) integer(i4b) If:

- TREND2=+1 The mean of the time segment is removed before computing the spectrum on this segment.
- TREND2=+2 The drift from the time segment is removed before computing the spectrum on this segment.
- TREND2=+3 The least-squares line from the time segment is removed before computing the spectrum on this segment.

For other values of TREND2 nothing is done before estimating the power spectrum on each segment.

The default is TREND2=0, e.g. nothing is done before estimating the power spectrum on each segment.

WIN (INPUT, OPTIONAL) integer(i4b) On entry, this argument specify the data window used in the computations of the power spectrum. If:

- WIN=+1 The Bartlett window is used
- WIN=+2 The square window is used
- WIN=+3 The Welch window is used
- WIN=+4 The Hann window is used
- WIN=+5 The Hamming window is used
- WIN=+6 A split-cosine-bell window is used

The default is WIN=3, e.g. the Welch window is used.

TAPERP (INPUT, OPTIONAL) real(stnd) The total percentage of the data to be tapered if WIN=6. TAPERP must be greater than zero and less or equal to one, otherwise the default value is used.

The default is 0.2 .

L0 (INPUT, OPTIONAL) integer(i4b) The number of zeros added to each time segment in order to obtain more finely spaced spectral estimates. L+L0 must be a positive even integer.

The default is L0=0, e.g. no zeros are added to each time segment.

PROBTEST (INPUT, OPTIONAL) real(stnd) On entry, a probability. PROBTEST is the critical probability which is used to determine the lower and upper confidence limit factors (e.g. the optional arguments CONLWR and CONUPR).

PROBTEST must verify: 0. < P < 1.

The default is 0.05 .

Further Details

After removing the mean or the trend from the time series (e.g. TREND=1,2,3), the series are padded with zero on the right such that the length of the resulting time series is evenly divisible by L (a positive even integer). The length, N, of this resulting time series is the first integer greater than or equal to size(MAT,2) which is evenly divisible by L. If size(MAT,2) is not evenly divisible by L, N is equal to size(MAT,2)+L-mod(size(MAT,2),L).

Optionally, the mean or the trend may also be removed from each time segment (e.g. TREND2=1,2,3). Optionally, zeros may be added to each time segment (e.g. the optional arguemnt L0) if more finely spaced spectral esimates are desired.

The stability of the PSD estimates depends on the averaging process. That is, the greater the number of segments (N/L if OVERLAP=false and (2N/L)-1 if OVERLAP=true), the more stable the resulting PSD estimates.

Optionally, theses PSD estimates may then be smoothed again in the frequency domain by modified Daniell filters (e.g. if argument NSMOOTH is used).

For definitions, more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York.

(2) **Welch, P.D., 1967:** The use of Fast Fourier Transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. IEEE trans. on audio and electroacoustics, Vol. Au-15, 2, 70-73.

(3) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford.

6.27.61 subroutine cross_spctrm2 (vec, vec2, l, psvec, psvec2, phase, coher, freq, edof, bandwidth, conlwr, conupr, testcoher, ampli, co_spect, quad_spect, prob_coher, initfft, overlap, normpsd, nsmooth, trend, trend2, win, taperp, l0, probtest)

Purpose

Subroutine CROSS_SPCTRM2 computes Fast Fourier Transform (FFT) estimates of the power and cross spectra of two real time series.

The Power Spectral Density (PSD) and Cross Spectral Density (CSD) estimates are returned in units which are the square of the data (if NORMPSD=false) or in spectral density units (if NORMPSD=true).

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the first real time series for which the power and cross spectra must be estimated. If TREND=1, 2 or 3, VEC is used as workspace and is transformed.

Size(VEC) must be greater or equal to 4.

VEC2 (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the second real time series for which the power and cross spectra must be estimated. If TREND=1, 2 or 3, VEC2 is used as workspace and is transformed.

VEC2 must verify: size(VEC2) = size(VEC).

L (INPUT) integer(i4b) On entry, an integer used to segment the time series. L is the length of the segments. L must be a positive even integer, less or equal to size(VEC), but greater or equal to 4.

Spectral computations are at (L/2)+1 frequencies if the optional argument L0 is absent and are at ((L+L0)/2)+1 frequencies if L0 is present (L0 is the number of zeros added to each segment).

Suggested values for L+L0 are 16, 32, 64 or 128 (e.g. an integer power of two, in order to speed the computations).

PSVEC (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the Power Spectral Density (PSD) estimates of VEC.

PSVEC must verify: size(PSVEC) = ((L+L0)/2) + 1 .

PSVEC2 (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the Power Spectral Density (PSD) estimates of VEC2.

PSVEC2 must verify: size(PSVEC2) = ((L+L0)/2) + 1 .

PHASE (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the phase of the cross spectrum, given in fractions of a circle (e.g. on the closed interval (0,1)).

PHASE must verify: size(PHASE) = ((L+L0)/2) + 1 .

COHER (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the squared coherency estimates for all frequencies.

COHER must verify: size(COHER) = ((L+L0)/2) + 1 .

FREQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the frequencies at which the spectral quantities are calculated in cycles per unit of time.

The spectral estimates are taken at frequencies (i-1)/(L+L0) for i=1,2, . . . , ((L+L0)/2 + 1).

FREQ must verify: size(FREQ) = (L+L0)/2 + 1 .

EDOF (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the equivalent number of degrees of freedom of the power spectrum estimates.

EDOF must verify: size(EDOF) = ((L+L0)/2) + 1 .

BANDWIDTH (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the bandwidth of the power spectrum estimates.

BANDWIDTH must verify: size(BANDWIDTH) = ((L+L0)/2) + 1 .

CONLWR (OUTPUT, OPTIONAL) real(stnd), dimension(:)

CONUPR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On output, these arguments specify the lower and upper (1-PROBTEST) * 100% confidence limit factors, respectively. Multiply the PSD estimates (e.g. the PSVEC(:) and PSVEC2(:) arguments) by these constants to get the lower and upper limits of a (1-PROBTEST) * 100% confidence interval for the PSD estimates.

CONLWR must verify: size(CONLWR) = ((L+L0)/2) + 1 .

CONUPR must verify: size(CONUPR) = ((L+L0)/2) + 1 .

TESTCOHER (OUTPUT, OPTIONAL) real(stnd), dimension(:) On output, this argument specifies the critical value for testing the null hypothesis that the squared coherency is zero at the PROBTEST * 100% significance level (e.g. elements of COHER(:) less than TESTCOHER(:) should be regarded as not significantly different from zero at the PROBTEST * 100% significance level).

TESTCOHER must verify: size(TESTCOHER) = ((L+L0)/2) + 1 .

AMPLI (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the cross-amplitude spectrum.

AMPLI must verify: size(AMPLI) = ((L+L0)/2) + 1 .

CO_SPECT (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the co-spectrum (e.g. the real part of cross-spectrum).

CO_SPECT must verify: size(CO_SPECT) = ((L+L0)/2) + 1 .

QUAD_SPECT (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the quadrature spectrum (e.g. the imaginary part of cross-spectrum with a minus sign).

QUAD_SPECT must verify: size(QUAD_SPECT) = ((L+L0)/2) + 1 .

PROB_COHER (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the probabilities that the computed sample squared coherencies came from an ergodic stationary bivariate process with (corresponding) squared coherencies equal to zero.

PROB_COHER must verify: size(PROB_COHER) = ((L+L0)/2)+1 .

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITFFT = false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine CROSS_SPCTRM2 in order to sets up constants and functions for use by subroutine FFT which is called inside subroutine CROSS_SPCTRM2. This call to INITFFT must have the following form:

  call init_fft((L+L0)/2)

- INITFFT = true, the call to INIT_FFT is done inside subroutine CROSS_SPCTRM2 and a call to END_FFT is also done before leaving subroutine CROSS_SPCTRM2.

The default is INITFFT=true .

OVERLAP (INPUT, OPTIONAL) logical(lgl) If:

- OVERLAP = false, the subroutine segments the data without any overlapping.
- OVERLAP=true, the subroutine overlaps the segments by one half of their length (which is equal to L).

In both cases, zeros are eventually added to each segment (if argument L0 is present) and each segment will be FFT'd, and the resulting periodograms will averaged together to obtain a Power Spectrum Density estimate at the ((L+L0)/2)+1 frequencies.

The default is OVERLAP=false .

NORMPSD (INPUT, OPTIONAL) logical(lgl) On entry, if:

- NORMPSD = true, the power and cross spectra estimates are normalized in such a way that the total area under the power spectrum is equal to the variance of the time series VEC and VEC2.
- NORMPSD = false, the sum of the PSD estimates (e.g. sum(PSVEC(2:)) and sum(PSVEC2(2:))) is equal to the variance of the corresponding time series.

The default is NORMPSD=true .

NSMOOTH (INPUT, OPTIONAL) integer(i4b) if NSMOOTH is used, the PSD and CSD estimates are computed by smoothing the periodogram with Daniell weights (e.g. a simple moving average).

On entry, NSMOOTH gives the length of the Daniell filter to be applied.

Setting NSMOOTH=0 on entry is equivalent to omit the optional argument NSMOOTH. Otherwise, NSMOOTH must be odd, greater than 2 and less or equal to (L+L0)/2+1 .

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+1 The mean of the two time series is removed before computing the spectra
- TREND=+2 The drift from the two time series is removed before computing the spectra
- TREND=+3 The least-squares line from the two time series is removed before computing the spectra.

For other values of TREND nothing is done before estimating the power and cross spectra. The default is TREND=1, e.g. the means of the time series are removed before the computations.

TREND2 (INPUT, OPTIONAL) integer(i4b) If:

- TREND2=+1 The mean of the time segment is removed before computing the cross-spectrum on this segment.
- TREND2=+2 The drift from the time segment is removed before computing the cross-spectrum on this segment.
- TREND2=+3 The least-squares line from the time segment is removed before computing the cross-spectrum on this segment.

For other values of TREND2 nothing is done before estimating the cross-spectrum on each segment.

The default is TREND2=0, e.g. nothing is done before estimating the power spectrum on each segment.

WIN (INPUT, OPTIONAL) integer(i4b) On entry, this argument specify the data window used in the computations of the power and cross spectra. If:

- WIN=+1 The Bartlett window is used
- WIN=+2 The square window is used
- WIN=+3 The Welch window is used
- WIN=+4 The Hann window is used
- WIN=+5 The Hamming window is used
- WIN=+6 A split-cosine-bell window is used

The default is WIN=3, e.g. the Welch window is used.

TAPERP (INPUT, OPTIONAL) real(stnd) The total percentage of the data to be tapered if WIN=6. TAPERP must be greater than zero and less or equal to one, otherwise the default value is used.

The default is 0.2 .

L0 (INPUT, OPTIONAL) integer(i4b) The number of zeros added to each time segment in order to obtain more finely spaced spectral estimates. L+L0 must be a positive even integer.

The default is L0=0, e.g. no zeros are added to each time segment.

PROBTEST (INPUT, OPTIONAL) real(stnd) On entry, a probability. PROBTEST is the critical probability which is used to determine the lower and upper confidence limit factors (e.g. the optional arguments CONLWR and CONUPR) and the critical value for testing the null hypothesis that the squared coherency is zero (e.g. the TESTCOHER optional argument).

PROBTEST must verify: 0. < P < 1.

The default is 0.05 .

Further Details

After removing the mean or the trend from the two time series (e.g. TREND=1,2,3), the series are padded with zero on the right such that the length of the resulting two time series is evenly divisible by L (a positive even integer). The length, N, of these resulting time series is the first integer greater than or equal to size(VEC) which is evenly divisible by L. If size(VEC) is not evenly divisible by L, N is equal to size(VEC)+L-mod(size(VEC),L).

Optionally, the mean or the trend may also be removed from each time segment (e.g. TREND2=1,2,3). Optionally, zeros may be added to each time segment (e.g. the optional arguemnt L0) if more finely spaced spectral esimates are desired.

The stability of the power and cross spectra estimates depends on the averaging process. That is, the greater the number of segments (N/L if OVERLAP=false and (2N/L)-1 if OVERLAP=true), the more stable the resulting power and cross spectra estimates.

Optionally, these power and cross spectra estimates may then be smoothed again in the frequency domain by modified Daniell filters (e.g. if argument NSMOOTH is used).

For definitions, more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York.

(2) **Welch, P.D., 1967:** The use of Fast Fourier Transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. IEEE trans. on audio and electroacoustics, Vol. Au-15, 2, 70-73.

(3) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford.

6.27.62 subroutine cross_spctrm2 (vec, mat, l, psvec, psmat, phase, coher, freq, edof, bandwidth, conlwr, conupr, testcoher, ampli, co_spect, quad_spect, prob_coher, initfft, overlap, normpsd, nsmooth, trend, trend2, win, taperp, l0, probtest)

Purpose

Subroutine CROSS_SPCTRM2 computes Fast Fourier Transform (FFT) estimates of the power and cross spectra of the real time series, VEC, and the multi-channel real time series MAT.

The Power Spectral Density (PSD) and Cross Spectral Density (CSD) estimates are returned in units which are the square of the data (if NORMPSD=false) or in spectral density units (if NORMPSD=true).

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the real time series for which the power and cross spectra must be estimated. If TREND=1, 2 or 3, VEC is used as workspace and is transformed.

Size(VEC) must be greater or equal to 4.

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the multi-channel real time series for which the power and cross spectra must be estimated. Each row of MAT is a real time series. If TREND=1, 2 or 3, MAT is used as workspace and is transformed.

The shape of MAT must verify: size(MAT,2) = size(VEC).

L (INPUT) integer(i4b) On entry, an integer used to segment the time series. L is the length of the segments. L must be a positive even integer, less or equal to size(VEC), but greater or equal to 4.

Spectral computations are at (L/2)+1 frequencies if the optional argument L0 is absent and are at ((L+L0)/2)+1 frequencies if L0 is present (L0 is the number of zeros added to each segment).

Suggested values for L+L0 are 16, 32, 64 or 128 (e.g. an integer power of two, in order to speed the computations).

PSVEC (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the Power Spectral Density (PSD) estimates of VEC.

PSVEC must verify: size(PSVEC) = ((L+L0)/2) + 1 .

PSMAT (OUTPUT) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and ((L+L0)/2) + 1 columns containing the Power Spectral Density (PSD) estimates of each row of MAT.

The shape of PSMAT must verify:

- size(PSMAT,1) = size(MAT,1) ;
- size(PSMAT,2) = ((L+L0)/2) + 1 .

PHASE (OUTPUT) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and ((L+L0)/2) + 1 columns containing the phase of the cross spectrum, given in fractions of a circle (e.g. on the closed interval (0,1)).

The shape of PHASE must verify:

- size(PHASE,1) = size(MAT,1) ;
- size(PHASE,2) = ((L+L0)/2) + 1 .

COHER (OUTPUT) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and ((L+L0)/2) + 1 columns containing the squared coherency estimates for all frequencies.

The shape of COHER must verify:

- size(COHER,1) = size(MAT,1) ;
- size(COHER,2) = ((L+L0)/2) + 1 .

FREQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the frequencies at which the spectral quantities are calculated in cycles per unit of time.

The spectral estimates are taken at frequencies (i-1)/(L+L0) for i=1,2, . . . , ((L+L0)/2 + 1).

FREQ must verify: size(FREQ) = (L+L0)/2 + 1 .

EDOF (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the equivalent number of degrees of freedom of the power spectrum estimates.

EDOF must verify: size(EDOF) = ((L+L0)/2) + 1 .

BANDWIDTH (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, the bandwidth of the power spectrum estimates.

BANDWIDTH must verify: size(BANDWIDTH) = ((L+L0)/2) + 1 .

CONLWR (OUTPUT, OPTIONAL) real(stnd), dimension(:)

CONUPR (OUTPUT, OPTIONAL) real(stnd), dimension(:) On output, these arguments specify the lower and upper (1-PROBTEST) * 100% confidence limit factors, respectively. Multiply the PSD estimates (e.g. the PSVEC(:) and PSMAT(:,:) arguments) by these constants to get the lower and upper limits of a (1-PROBTEST) * 100% confidence interval for the PSD estimates.

CONLWR must verify: size(CONLWR) = ((L+L0)/2) + 1 .

CONUPR must verify: size(CONUPR) = ((L+L0)/2) + 1 .

TESTCOHER (OUTPUT, OPTIONAL) real(stnd), dimension(:) On output, this argument specifies the critical value for testing the null hypothesis that the squared coherency is zero at the PROBTEST * 100% significance level (e.g. elements of COHER(:,:) less than TESTCOHER(:) should be regarded as not significantly different from zero at the PROBTEST * 100% significance level).

TESTCOHER must verify: size(TESTCOHER) = ((L+L0)/2) + 1 .

AMPLI (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and ((L+L0)/2) + 1 columns containing the cross-amplitude spectra.

The shape of AMPLI must verify:

- size(AMPLI,1) = size(MAT,1) ;
- size(AMPLI,2) = ((L+L0)/2) + 1 .

CO_SPECT (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and ((L+L0)/2) + 1 columns containing the co-spectra (e.g. the real part of cross-spectra).

The shape of CO_SPECT must verify:

- size(CO_SPECT,1) = size(MAT,1) ;
- size(CO_SPECT,2) = ((L+L0)/2) + 1 .

QUAD_SPECT (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and ((L+L0)/2) + 1 columns containing the quadrature spectrum (e.g. the imaginary part of cross-spectrum with a minus sign).

The shape of QUAD_SPECT must verify:

- size(QUAD_SPECT,1) = size(MAT,1) ;
- size(QUAD_SPECT,2) = ((L+L0)/2) + 1 .

PROB_COHER (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and ((L+L0)/2) + 1 columns containing the probabilities that the computed sample squared coherencies came from an ergodic stationary bivariate process with (corresponding) squared coherencies equal to zero.

The shape of PROB_COHER must verify:

size(PROB_COHER,1) = size(MAT,1) ; size(PROB_COHER,2) = ((L+L0)/2) + 1 .

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITFFT = false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine CROSS_SPCTRM2 in order to sets up constants and functions for use by subroutine FFT which is called inside subroutine CROSS_SPCTRM2. This call to INITFFT must have the following form:

  call init_fft((/ size(MAT,1), (L+L0)/2 /), dim=2_i4b)

- INITFFT is set to true, the call to INIT_FFT is done inside subroutine CROSS_SPCTRM2 and a call to END_FFT is also done before leaving subroutine CROSS_SPCTRM2.

The default is INITFFT=true .

OVERLAP (INPUT, OPTIONAL) logical(lgl) If:

- OVERLAP = false, the subroutine segments the data without any overlapping.
- OVERLAP=true, the subroutine overlaps the segments by one half of their length (which is equal to L).

In both cases, zeros are eventually added to each segment (if argument L0 is present) and each segment will be FFT'd, and the resulting periodograms will averaged together to obtain a Power Spectrum Density estimate at the ((L+L0)/2)+1 frequencies.

The default is OVERLAP=false .

NORMPSD (INPUT, OPTIONAL) logical(lgl) On entry, if:

- NORMPSD = true, the power and cross spectra estimates are normalized in such a way that the total area under the power spectra is equal to the variance of the time series contained in VEC and in each row of MAT.
- NORMPSD is set to false, the sum of the PSD estimates (e.g. sum(PSVEC(2:)) and sum(PSMAT(:,2:),dim=2)) is equal to the variance of the corresponding time series.

The default is NORMPSD=true .

NSMOOTH (INPUT, OPTIONAL) integer(i4b) If NSMOOTH is used, the PSD estimates are computed by smoothing the periodogram with Daniell weights (e.g. a simple moving average).

On entry, NSMOOTH gives the length of the Daniell filter to be applied.

Setting NSMOOTH=0 on entry is equivalent to omit the optional argument NSMOOTH. Otherwise, NSMOOTH must be odd, greater than 2 and less or equal to (L+L0)/2+1 .

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+1 The means of the time series are removed before computing the spectra
- TREND=+2 The drifts from time series are removed before computing the spectra
- TREND=+3 The least-squares lines from time series are removed before computing the spectra.

For other values of TREND nothing is done before estimating the power and cross spectra. The default is TREND=1, e.g. the means of the time series are removed before the computations.

TREND2 (INPUT, OPTIONAL) integer(i4b) If:

- TREND2=+1 The mean of the time segment is removed before computing the cross-spectrum on this segment.
- TREND2=+2 The drift from the time segment is removed before computing the cross-spectrum on this segment.
- TREND2=+3 The least-squares line from the time segment is removed before computing the cross-spectrum on this segment.

For other values of TREND2 nothing is done before estimating the cross-spectrum on each segment.

The default is TREND2=0, e.g. nothing is done before estimating the power spectrum on each segment.

WIN (INPUT, OPTIONAL) integer(i4b) On entry, this argument specify the data window used in the computations of the power and cross spectra. If:

- WIN=+1 The Bartlett window is used
- WIN=+2 The square window is used
- WIN=+3 The Welch window is used
- WIN=+4 The Hann window is used
- WIN=+5 The Hamming window is used
- WIN=+6 A split-cosine-bell window is used

The default is WIN=3, e.g. the Welch window is used.

TAPERP (INPUT, OPTIONAL) real(stnd) The total percentage of the data to be tapered if WIN=6. TAPERP must be greater than zero and less or equal to one, otherwise the default value is used.

The default is 0.2 .

L0 (INPUT, OPTIONAL) integer(i4b) The number of zeros added to each time segment in order to obtain more finely spaced spectral estimates. L+L0 must be a positive even integer.

The default is L0=0, e.g. no zeros are added to each time segment.

PROBTEST (INPUT, OPTIONAL) real(stnd) On entry, a probability. PROBTEST is the critical probability which is used to determine the lower and upper confidence limit factors (e.g. the optional arguments CONLWR and CONUPR) and the critical value for testing the null hypothesis that the squared coherency is zero (e.g. the TESTCOHER optional argument).

PROBTEST must verify 0. < P < 1.

The default is 0.05 .

Further Details

After removing the mean or the trend from the time series (e.g. TREND=1,2,3), the series are padded with zero on the right such that the length of the resulting time series is evenly divisible by L (a positive even integer). The length, N, of these resulting time series is the first integer greater than or equal to size(VEC) which is evenly divisible by L. If size(VEC) is not evenly divisible by L, N is equal to size(VEC)+L-mod(size(VEC),L).

Optionally, the mean or the trend may also be removed from each time segment (e.g. TREND2=1,2,3). Optionally, zeros may be added to each time segment (e.g. the optional arguemnt L0) if more finely spaced spectral esimates are desired.

The stability of the power and cross spectra estimates depends on the averaging process. That is, the greater the number of segments (N/L if OVERLAP=false and (2N/L)-1 if OVERLAP=true), the more stable the resulting power and cross spectra estimates.

Optionally, these power and cross spectra estimates may then be smoothed again in the frequency domain by modified Daniell filters (e.g. if argument NSMOOTH is used).

For definitions, more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York.

(2) **Welch, P.D., 1967:** The use of Fast Fourier Transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. IEEE trans. on audio and electroacoustics, Vol. Au-15, 2, 70-73.

(3) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford.

6.27.63 subroutine power_spectrum (vec, psvec, freq, fftvec, edof, bandwidth, conlwr, conupr, initfft, normpsd, smooth_param, trend, win, taperp, probtest)

Purpose

Subroutine POWER_SPECTRUM computes a Fast Fourier Transform (FFT) estimate of the power spectrum of a real time series, VEC. The real valued sequence VEC must be of even length.

The Power Spectral Density (PSD) estimates are returned in units which are the square of the data (if NORMPSD=false) or in spectral density units (if NORMPSD=true).

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the real time series for which the power spectrum must be estimated. If WIN/=2 or TREND=1, 2 or 3, VEC is used as workspace and is transformed.

Size(VEC) must be an even (positive) integer greater or equal to 4.

PSVEC (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the Power Spectral Density (PSD) estimates of VEC.

PSVEC must verify: size(PSVEC) = size(VEC)/2 + 1 .

FREQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the frequencies at which the spectral quantities are calculated in cycles per unit of time.

The spectral estimates are taken at frequencies (i-1)/size(VEC) for i=1,2, . . . , (size(VEC)/2 + 1).

FREQ must verify: size(FREQ) = size(VEC)/2 + 1 .

FFTVEC (OUTPUT, OPTIONAL) complex(stnd), dimension(:) On exit, a complex vector of length (size(VEC)/2)+1 containing the Fast Fourier Transform of the product of the (detrended, e.g. the TREND argument) real time series VEC with the choosen window function (e.g. The WIN argument).

FFTVEC must verify: size(FFTVEC) = size(VEC)/2 + 1 .

EDOF (OUTPUT, OPTIONAL) real(stnd) On exit, the equivalent number of degrees of freedom of the power spectrum estimates.

BANDWIDTH (OUTPUT, OPTIONAL) real(stnd) On exit, the bandwidth of the power spectrum estimates.

CONLWR (OUTPUT, OPTIONAL) real(stnd)

CONUPR (OUTPUT, OPTIONAL) real(stnd) On output, these arguments specify the lower and upper (1-PROBTEST) * 100% confidence limit factors, respectively. Multiply the PSD estimates (e.g. the PSVEC(:) argument) by these constants to get the lower and upper limits of a (1-PROBTEST) * 100% confidence interval for the PSD estimates.

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITFFT = false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine POWER_SPECTRUM in order to sets up constants and functions for use by subroutine FFT which is called inside subroutine POWER_SPECTRUM. This call to INITFFT must have the following form:

  call init_fft(size(VEC)/2)

- INITFFT = true, the call to INIT_FFT is done inside subroutine POWER_SPECTRUM and a call to END_FFT is also done before leaving subroutine POWER_SPECTRUM.

The default is INITFFT=true .

NORMPSD (INPUT, OPTIONAL) logical(lgl) On entry, if:

- NORMPSD = true, the PSD estimates are normalized in such a way that the total area under the power spectrum is equal to the variance of the time series VEC.
- NORMPSD is set to false, the sum of the PSD estimates (e.g. sum(PSVEC(2:)) is equal to the variance of the time series.

The default is NORMPSD=true .

SMOOTH_PARAM (INPUT, OPTIONAL) integer(i4b), dimension(:) if SMOOTH_PARAM is used, the PSD estimates are computed by repeated smoothing of the periodogram with modified Daniell weights.

On entry, SMOOTH_PARAM(:) gives the array of the half-lengths of the modified Daniell filters to be applied.

All the values in SMOOTH_PARAM(:) must be greater than 0 and less than size(VEC)/2+1 .

Size(SMOOTH_PARAM) must be greater or equal to 1.

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+1 The mean of the time series is removed before computing the spectrum
- TREND=+2 The drift from the time series is removed before computing the spectrum by using the formula: drift = (VEC(size(VEC)) - VEC(1))/(size(VEC) - 1)
- TREND=+3 The least-squares line from the time series is removed before computing the spectrum.

For other values of TREND nothing is done before estimating the power spectrum.

The default is TREND=1, e.g. the mean of the time series is removed before the computations.

WIN (INPUT, OPTIONAL) integer(i4b) On entry, this argument specify the data window used in the computations of the power spectrum. If:

- WIN=+1 The Bartlett window is used
- WIN=+2 The square window is used
- WIN=+3 The Welch window is used
- WIN=+4 The Hann window is used
- WIN=+5 The Hamming window is used
- WIN=+6 A split-cosine-bell window is used

The default is WIN=3, e.g. the Welch window is used.

TAPERP (INPUT, OPTIONAL) real(stnd) The total percentage of the data to be tapered if WIN=6. TAPERP must be greater than zero and less or equal to one, otherwise the default value is used.

The default is 0.2 .

PROBTEST (INPUT, OPTIONAL) real(stnd) On entry, a probability. PROBTEST is the critical probability which is used to determine the lower and upper confidence limit factors (e.g. the optional arguments CONLWR and CONUPR).

PROBTEST must verify: 0. < P < 1.

The default is 0.05 .

Further Details

After removing the mean or the trend from the time series (e.g. TREND=1,2,3), the selected data window (e.g. WIN=1,2,3,4,5,6) is applied to the time series and the PSD estimates are computed by the FFT of this transformed time series. Optionally, theses PSD estimates may then be smoothed in the frequency domain by modified Daniell filters (e.g. if SMOOTH_PARAM is used).

The computed equivalent number of degrees of freedom and bandwidth must be divided by two for the zero and Nyquist frequencies.

Furthermore, the computed equivalent number of degrees of freedom, bandwidth, lower and upper (1-PROBTEST) * 100% confidence limit factors are not right near the zero and Nyquist frequencies if the PSD estimates have been smoothed by modified Daniell filters. The reason is that POWER_SPECTRUM assumes that smoothing involves averaging independent frequency ordinates. This is true except near the zero and Nyquist frequencies where an average may contain contributions from negative frequencies, which are identical to and hence not independent of positive frequency spectral values. Thus, the number of degrees of freedom in PSD estimates near the 0 and Nyquist frequencies are as little as half the number of degrees of freedom of the spectral estimates away from these frequency extremes if the optional argument SMOOTH_PARAM is used.

If the optional argument SMOOTH_PARAM is used, the computed equivalent number of degrees of freedom, bandwidth, lower and upper (1-PROBTEST) * 100% confidence limit factors are right for PSD estimates at frequencies

(i-1)/Size(VEC) for i= (nparam+1)/2 + 1 to (Size(VEC) - nparam + 1)/2

where nparam = 2 * (2+sum(SMOOTH_PARAM(:)))- 1, (e.g. for frequencies i/Size(VEC) for i = (nparam+1)/2, . . . , (Size(VEC) - nparam - 1)/2).

For definitions, more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York.

(2) **Welch, P.D., 1967:** The use of Fast Fourier Transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. IEEE trans. on audio and electroacoustics, Vol. Au-15, 2, 70-73.

(3) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford.

6.27.64 subroutine power_spectrum (mat, psmat, freq, fftmat, edof, bandwidth, conlwr, conupr, initfft, normpsd, smooth_param, trend, win, taperp, probtest)

Purpose

Subroutine POWER_SPECTRUM computes a Fast Fourier Transform (FFT) estimate of the power spectra of the rows of the real matrix, MAT. size(MAT,2) must be of even length.

The Power Spectral Density (PSD) estimates are returned in units which are the square of the data (if NORMPSD=false) or in spectral density units (if NORMPSD=true).

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real time series for which power spectra must be estimated. Each row of MAT is a real time series. If WIN/=2 or TREND=1, 2 or 3, MAT is used as workspace and is transformed.

Size(MAT,2) must be an even (positive) integer greater or equal to 4.

PSMAT (OUTPUT) real(stnd), dimension(:,:) On exit, a real matrix containing the Power Spectral Density (PSD) estimates for each row of the real matrix MAT.

The shape of PSMAT must verify:

- size(PSMAT,1) = size(MAT,1) ;
- size(PSMAT,2) = size(MAT,2)/2 + 1 .

FREQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length (size(MAT,2)/2)+1 containing the frequencies at which the spectral quantities are calculated in cycles per unit of time.

The spectral estimates are taken at frequencies (i-1)/size(VEC) for i=1,2, . . . , (size(MAT,2)/2 + 1).

FREQ must verify: size(FREQ) = size(MAT,2)/2 + 1 .

FFTMAT (OUTPUT, OPTIONAL) complex(stnd), dimension(:,:) On exit, a complex matrix containing the Fast Fourier Transform of the product of the (detrended, e.g. the TREND argument) real time series in each row of MAT with the choosen window function (e.g. The WIN argument).

The shape of FFTMAT must verify:

- size(FFTMAT,1) = size(MAT,1) ;
- size(FFTMAT,2) = size(MAT,2)/2 + 1 .

EDOF (OUTPUT, OPTIONAL) real(stnd) On exit, the equivalent number of degrees of freedom of the power spectrum estimates.

BANDWIDTH (OUTPUT, OPTIONAL) real(stnd) On exit, the bandwidth of the power spectrum estimates.

CONLWR (OUTPUT, OPTIONAL) real(stnd)

CONUPR (OUTPUT, OPTIONAL) real(stnd) On output, these arguments specify the lower and upper (1-PROBTEST) * 100% confidence limit factors, respectively. Multiply the PSD estimates (e.g. the PSMAT(:,:) argument) by these constants to get the lower and upper limits of a (1-PROBTEST) * 100% confidence interval for the PSD estimates.

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITFFT = false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine POWER_SPECTRUM in order to sets up constants and functions for use by subroutine FFT which is called inside subroutine POWER_SPECTRUM. This call to INITFFT must have the following form:

  call init_fft((/ size(MAT,1), size(MAT,2)/2 /), dim=2_i4b)

- INITFFT = true, the call to INIT_FFT is done inside subroutine POWER_SPECTRUM and a call to END_FFT is also done before leaving subroutine POWER_SPECTRUM

The default is INITFFT=true .

NORMPSD (INPUT, OPTIONAL) logical(lgl) On entry, if:

- NORMPSD = true, the PSD estimates are normalized in such a way that the total area under the power spectrum is equal to the variance of the time series MAT.
- NORMPSD = false, the sum of the PSD estimates for each row of MAT (e.g. sum(PSMAT(:,2:), dim=2) is equal to the variance of the corresponding time series.

The default is NORMPSD=true .

SMOOTH_PARAM (INPUT, OPTIONAL) integer(i4b), dimension(:) if SMOOTH_PARAM is used, the PSD estimates are computed by repeated smoothing of the periodogram with modified Daniell weights.

On entry, SMOOTH_PARAM(:) gives the array of the half-lengths of the modified Daniell filters to be applied.

All the values in SMOOTH_PARAM(:) must be greater than 0 and less than size(MAT,2)/2 + 1 .

Size(SMOOTH_PARAM) must be greater or equal to 1.

TREND (INPUT, OPTIONAL) integer(i4b) If

- TREND=+1 The means of the time series are removed before computing the spectra
- TREND=+2 The drifts from the time series are removed before computing the spectra by using the formula: drift(i) = (MAT(i,size(MAT,2)) - MAT(i,1))/(size(MAT,2) - 1)
- TREND=+3 The least-squares lines from the time series are removed before computing the spectra.

For other values of TREND nothing is done before estimating the power spectra.

The default is TREND=1, e.g. the means of the time series are removed before the computations.

WIN (INPUT, OPTIONAL) integer(i4b) On entry, this argument specify the data window used in the computations of the power spectrum. If:

- WIN=+1 The Bartlett window is used
- WIN=+2 The square window is used
- WIN=+3 The Welch window is used
- WIN=+4 The Hann window is used
- WIN=+5 The Hamming window is used
- WIN=+6 A split-cosine-bell window is used

The default is WIN=3, e.g. the Welch window is used.

TAPERP (INPUT, OPTIONAL) real(stnd) The total percentage of the data to be tapered if WIN=6. TAPERP must be greater than zero and less or equal to one, otherwise the default value is used.

The default is 0.2 .

PROBTEST (INPUT, OPTIONAL) real(stnd) On entry, a probability. PROBTEST is the critical probability which is used to determine the lower and upper confidence limit factors (e.g. the optional arguments CONLWR and CONUPR).

PROBTEST must verify: 0. < P < 1.

The default is 0.05 .

Further Details

After removing the mean or the trend from the time series (e.g. TREND=1,2,3), the selected data window (e.g. WIN=1,2,3,4,5,6) is applied to the time series and the PSD estimates are computed by the FFT of these transformed time series. Optionally, theses PSD estimates may then be smoothed in the frequency domain by modified Daniell filters (e.g. if SMOOTH_PARAM is used).

The computed equivalent number of degrees of freedom and bandwidth must be divided by two for the zero and Nyquist frequencies.

Furthermore, the computed equivalent number of degrees of freedom, bandwidth, lower and upper (1-PROBTEST) * 100% confidence limit factors are not right near the zero and Nyquist frequencies if the PSD estimates have been smoothed by modified Daniell filters. The reason is that POWER_SPECTRUM assumes that smoothing involves averaging independent frequency ordinates. This is true except near the zero and Nyquist frequencies where an average may contain contributions from negative frequencies, which are identical to and hence not independent of positive frequency spectral values. Thus, the number of degrees of freedom in PSD estimates near the 0 and Nyquist frequencies are as little as half the number of degrees of freedom of the spectral estimates away from these frequency extremes if the optional argument SMOOTH_PARAM is used.

If the optional argument SMOOTH_PARAM is used, the computed equivalent number of degrees of freedom, bandwidth, lower and upper (1-PROBTEST) * 100% confidence limit factors are right for PSD estimates at frequencies

(i-1)/Size(MAT,2) for i= (nparam+1)/2 + 1 to (Size(MAT,2) - nparam + 1)/2

where nparam = 2 * (2+sum(SMOOTH_PARAM(:)))- 1, (e.g. for frequencies i/Size(MAT,2) for i = (nparam+1)/2, . . . , (Size(MAT,2) - nparam - 1)/2).

For definitions, more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York.

(2) **Welch, P.D., 1967:** The use of Fast Fourier Transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. IEEE trans. on audio and electroacoustics, Vol. Au-15, 2, 70-73.

(3) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford.

6.27.65 subroutine cross_spectrum (vec, vec2, psvec, psvec2, phase, coher, freq, edof, bandwidth, conlwr, conupr, testcoher, ampli, co_spect, quad_spect, prob_coher, initfft, normpsd, smooth_param, trend, win, taperp, probtest)

Purpose

Subroutine CROSS_SPECTRUM computes Fast Fourier Transform (FFT) estimates of the power and cross spectra of two real time series, VEC and VEC2. The real valued sequences VEC and VEC2 must be of even length.

The Power Spectral Density (PSD) and Cross Spectral Density (CSD) estimates are returned in units which are the square of the data (if NORMPSD=false) or in spectral density units (if NORMPSD=true).

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the first real time series for which the power and cross spectra must be estimated. If WIN/=2 or TREND=1, 2 or 3, VEC is used as workspace and is transformed.

Size(VEC) must be an even (positive) integer greater or equal to 4.

VEC2 (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the second real time series for which the power and cross spectra must be estimated. If WIN/=2 or TREND=1, 2 or 3, VEC2 is used as workspace and is transformed.

VEC2 must verify: size(VEC2) = size(VEC).

PSVEC (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the Power Spectral Density (PSD) estimates of VEC.

PSVEC must verify: size(PSVEC) = size(VEC)/2 + 1 .

PSVEC2 (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length (size(VEC2)/2)+1 containing the Power Spectral Density (PSD) estimates of VEC2.

PSVEC2 must verify: size(PSVEC2) = size(VEC)/2 + 1 .

PHASE (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the phase of the cross spectrum, given in fractions of a circle (e.g. on the closed interval (0,1)).

PHASE must verify: size(PHASE) = size(VEC)/2 + 1 .

COHER (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the squared coherency estimates for all frequencies.

COHER must verify: size(COHER) = size(VEC)/2 + 1 .

FREQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the frequencies at which the spectral quantities are calculated in cycles per unit of time.

The spectral estimates are taken at frequencies (i-1)/size(VEC) for i=1,2, . . . , (size(VEC)/2 + 1).

FREQ must verify: size(FREQ) = size(VEC)/2 + 1 .

EDOF (OUTPUT, OPTIONAL) real(stnd) On exit, the equivalent number of degrees of freedom of the power and cross spectrum estimates.

BANDWIDTH (OUTPUT, OPTIONAL) real(stnd) On exit, the bandwidth of the power and cross spectrum estimates.

CONLWR (OUTPUT, OPTIONAL) real(stnd)

CONUPR (OUTPUT, OPTIONAL) real(stnd) On output, these arguments specify the lower and upper (1-PROBTEST) * 100% confidence limit factors, respectively. Multiply the PSD estimates (e.g. the PSVEC(:) and PSVEC2(:) arguments) by these constants to get the lower and upper limits of a (1-PROBTEST) * 100% confidence interval for the PSD estimates.

TESTCOHER (OUTPUT, OPTIONAL) real(stnd) On output, this argument specifies the critical value for testing the null hypothesis that the squared coherency is zero at the PROBTEST * 100% significance level (e.g. elements of COHER(:) less than TESTCOHER should be regarded as not significantly different from zero at the PROBTEST * 100% significance level).

AMPLI (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the cross-amplitude spectrum.

AMPLI must verify: size(AMPLI) = (size(VEC)/2) + 1 .

CO_SPECT (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the co-spectrum (e.g. the real part of cross-spectrum).

CO_SPECT must verify: size(CO_SPECT) = (size(VEC)/2) + 1 .

QUAD_SPECT (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the quadrature spectrum (e.g. the imaginary part of cross-spectrum with a minus sign).

QUAD_SPECT must verify: size(QUAD_SPECT) = (size(VEC)/2) + 1 .

PROB_COHER (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the probabilities that the computed sample squared coherencies came from an ergodic stationary bivariate process with (corresponding) squared coherencies equal to zero.

PROB_COHER must verify: size(PROB_COHER) = (size(VEC)/2) + 1 .

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITFFT = false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine CROSS_SPECTRUM in order to sets up constants and functions for use by subrou-

tine FFT which is called inside subroutine CROSS_SPECTRUM. This call to INITFFT must have the following form:

call init_fft(size(VEC)/2)

- INITFFT = true, the call to INIT_FFT is done inside subroutine CROSS_SPECTRUM and a call to END_FFT is also done before leaving subroutine CROSS_SPECTRUM.

The default is INITFFT=true .

NORMPSD (INPUT, OPTIONAL) logical(lgl) On entry, if:

- NORMPSD = true, the power and cross spectra estimates are normalized in such a way that the total area under the power spectrum is equal to the variance of the time series VEC and VEC2.
- NORMPSD = false, the sum of the PSD estimates (e.g. sum(PSVEC(2:)) and sum(PSVEC2(2:))) is equal to the variance of the corresponding time series.

The default is NORMPSD=true .

SMOOTH_PARAM (INPUT, OPTIONAL) integer(i4b), dimension(:) if SMOOTH_PARAM is used, the power and cross spectra estimates are computed by repeated smoothing of the periodograms and cross-periodogram with modified Daniell weights.

On entry, SMOOTH_PARAM(:) gives the array of the half-lengths of the modified Daniell filters to be applied.

All the values in SMOOTH_PARAM(:) must be greater than 0 and less than size(VEC)/2+1 .

Size(SMOOTH_PARAM) must be greater or equal to 1.

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+1 The mean of the two time series is removed before computing the power and cross spectra.
- TREND=+2 The drift from the two time series is removed before computing the power and cross spectra.
- TREND=+3 The least-squares line from the two time series is removed before computing the power and cross spectra.

For other values of TREND nothing is done before estimating the power and cross spectra.

The default is TREND=1, e.g. the means of the time series are removed before the computations.

WIN (INPUT, OPTIONAL) integer(i4b) On entry, this argument specify the data window used in the computations of the power and cross spectra. If

- WIN=+1 The Bartlett window is used
- WIN=+2 The square window is used
- WIN=+3 The Welch window is used
- WIN=+4 The Hann window is used
- WIN=+5 The Hamming window is used
- WIN=+6 A split-cosine-bell window is used

The default is WIN=3, e.g. the Welch window is used.

TAPERP (INPUT, OPTIONAL) real(stnd) The total percentage of the data to be tapered if WIN=6. TAPERP must be greater than zero and less or equal to one, otherwise the default value is used.

The default is 0.2 .

PROBTEST (INPUT, OPTIONAL) real(stnd) On entry, a probability. PROBTEST is the critical probability which is used to determine the lower and upper confidence limit factors (e.g. the optional arguments CONLWR and CONUPR) and the critical value for testing the null hypothesis that the squared coherency is zero (e.g. the TESTCOHER optional argument).

PROBTEST must verify: 0. < P < 1.

The default is 0.05 .

Further Details

After removing the mean or the trend from the time series (e.g. TREND=1,2,3), the selected data window (e.g. WIN=1,2,3,4,5,6) is applied to the time series and the PSD and CSD estimates are computed by the FFT of these transformed time series. Optionally, theses PSD and CSD estimates may then be smoothed in the frequency domain by modified Daniell filters (e.g. if argument SMOOTH_PARAM is used).

The computed equivalent number of degrees of freedom and bandwidth must be divided by two for the zero and Nyquist frequencies.

Furthermore, the computed equivalent number of degrees of freedom, bandwidth, lower and upper (1-PROBTEST) * 100% confidence limit factors and critical value for the squared coherency (e.g. arguments EDOF, BANDWIDTH, CONLWR, CONUPR and TESTCOHER) are not right near the zero and Nyquist frequencies if the PSD estimates have been smoothed by modified Daniell filters. The reason is that CROSS_SPECTRUM assumes that smoothing involves averaging independent frequency ordinates. This is true except near the zero and Nyquist frequencies where an average may contain contributions from negative frequencies, which are identical to and hence not independent of positive frequency spectral values. Thus, the number of degrees of freedom in PSD estimates near the 0 and Nyquist frequencies are as little as half the number of degrees of freedom of the spectral estimates away from these frequency extremes if the optional argument SMOOTH_PARAM is used.

If the optional argument SMOOTH_PARAM is used, the computed equivalent number of degrees of freedom, bandwidth, lower and upper (1-PROBTEST) * 100% confidence limit factors and critical value for the squared coherency are right for PSD estimates at frequencies

(i-1)/size(VEC) for i= (nparam+1)/2 + 1 to (size(VEC) - nparam + 1)/2

where nparam = 2 * (2+sum(SMOOTH_PARAM(:)))- 1, (e.g. for frequencies i/size(VEC) for i = (nparam+1)/2, . . . , (size(VEC)-nparam-1)/2) .

For definitions, more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York.

(2) **Welch, P.D., 1967:** The use of Fast Fourier Transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. IEEE trans. on audio and electroacoustics, Vol. Au-15, 2, 70-73.

(3) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford.

6.27.66 subroutine cross_spectrum (vec, mat, psvec, psmat, phase, coher, freq, edof, bandwidth, conlwr, conupr, testcoher, ampli, co_spect, quad_spect, prob_coher, initfft, normpsd, smooth_param, trend, win, taperp, probtest)

Purpose

Subroutine CROSS_SPECTRUM computes Fast Fourier Transform (FFT) estimates of the power and cross spectra of the real time series, VEC, and the multi-channel real time series MAT.

The Power Spectral Density (PSD) and Cross Spectral Density (CSD) estimates are returned in units which are the square of the data (if NORMPSD=false) or in spectral density units (if NORMPSD=true).

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the real time series for which the power and cross spectra must be estimated. If WIN/=2 or TREND=1, 2 or 3, VEC is used as workspace and is transformed.

Size(VEC) must be an even (positive) integer greater or equal to 4.

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the multi-channel real time series for which the power and cross spectra must be estimated. Each row of MAT is a real time series. If WIN/=2 or TREND=1, 2 or 3, MAR is used as workspace and is transformed.

The shape of MAT must verify: size(MAT,2) = size(VEC).

PSVEC (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the Power Spectral Density (PSD) estimates of VEC.

PSVEC must verify: size(PSVEC) = size(VEC)/2 + 1 .

PSMAT (OUTPUT) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and (size(VEC)/2)+1 columns containing the Power Spectral Density (PSD) estimates of each row of MAT.

The shape of PSMAT must verify:

- size(PSMAT,1) = size(MAT,1) ;
- size(PSMAT,2) = size(VEC)/2 + 1 .

PHASE (OUTPUT) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and (size(VEC)/2)+1 columns containing the phase of the cross spectrum, given in fractions of a circle (e.g. on the closed interval (0,1)).

The shape of PHASE must verify:

- size(PHASE,1) = size(MAT,1) ;
- size(PHASE,2) = size(VEC)/2 + 1 .

COHER (OUTPUT) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and (size(VEC)/2)+1 columns containing the squared coherency estimates for all frequencies.

The shape of COHER must verify:

- size(COHER,1) = size(MAT,1) ;
- size(COHER,2) = size(VEC)/2 + 1 .

FREQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length (size(VEC)/2)+1 containing the frequencies at which the spectral quantities are calculated in cycles per unit of time.

The spectral estimates are taken at frequencies (i-1)/size(VEC) for i=1,2, . . . , (size(VEC)/2 + 1).

FREQ must verify: size(FREQ) = size(VEC)/2 + 1 .

EDOF (OUTPUT, OPTIONAL) real(stnd) On exit, the equivalent number of degrees of freedom of the power and cross spectrum estimates.

BANDWIDTH (OUTPUT, OPTIONAL) real(stnd) On exit, the bandwidth of the power and cross spectrum estimates.

CONLWR (OUTPUT, OPTIONAL) real(stnd)

CONUPR (OUTPUT, OPTIONAL) real(stnd) On output, these arguments specify the lower and upper (1-PROBTEST) * 100% confidence limit factors, respectively. Multiply the PSD estimates (e.g. the PSVEC(:) and PSMAT(:,:) arguments) by these constants to get the lower and upper limits of a (1-PROBTEST) * 100% confidence interval for the PSD estimates.

TESTCOHER (OUTPUT, OPTIONAL) real(stnd) On output, this argument specifies the critical value for testing the null hypothesis that the squared coherency is zero at the PROBTEST * 100% significance level (e.g. elements of COHER(:,:) less than TESTCOHER should be regarded as not significantly different from zero at the PROBTEST * 100% significance level).

AMPLI (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and (size(VEC)/2)+1 columns containing the cross-amplitude spectra.

The shape of AMPLI must verify:

- size(AMPLI,1) = size(MAT,1) ;
- size(AMPLI,2) = (size(VEC)/2) + 1 .

CO_SPECT (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and (size(VEC)/2)+1 columns containing the co-spectra (e.g. the real part of cross-spectra).

The shape of CO_SPECT must verify:

- size(CO_SPECT,1) = size(MAT,1) ;
- size(CO_SPECT,2) = (size(VEC)/2) + 1 .

QUAD_SPECT (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and (size(VEC)/2)+1 columns containing the quadrature spectrum (e.g. the imaginary part of cross-spectrum with a minus sign).

The shape of QUAD_SPECT must verify:

- size(QUAD_SPECT,1) = size(MAT,1) ;
- size(QUAD_SPECT,2) = (size(VEC)/2) + 1 .

PROB_COHER (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and (size(VEC)/2)+1 columns containing the probabilities that the computed sample squared coherencies came from an ergodic stationary bivariate process with (corresponding) squared coherencies equal to zero.

The shape of PROB_COHER must verify:

- size(PROB_COHER,1) = size(MAT,1) ;
- size(PROB_COHER,2) = (size(VEC)/2) + 1 .

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITFFT = false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine CROSS_SPECTRUM in order to sets up constants and functions for use by subroutine FFT which is called inside subroutine CROSS_SPECTRUM. This call to INITFFT must have the following form:

call init_fft((/ size(MAT,1), size(MAT,2)/2 /), dim=2_i4b)

- INITFFT = true, the call to INIT_FFT is done inside subroutine CROSS_SPECTRUM and a call to END_FFT is also done before leaving subroutine CROSS_SPECTRUM.

The default is INITFFT=true .

NORMPSD (INPUT, OPTIONAL) logical(lgl) On entry, if:

- NORMPSD = true, the power and cross spectra estimates are normalized in such a way that the total area under the power spectra is equal to the variance of the time series contained in VEC and in each row of MAT.
- NORMPSD = false, the sum of the PSD estimates (e.g. sum(PSVEC(2:)) and sum(PSMAT(:,2:),dim=2)) is equal to the variance of the corresponding time series.

The default is NORMPSD=true .

SMOOTH_PARAM (INPUT, OPTIONAL) integer(i4b), dimension(:) if SMOOTH_PARAM is used, the power and cross spectra estimates are computed by repeated smoothing of the periodograms and cross-periodogram with modified Daniell weights.

On entry, SMOOTH_PARAM(:) gives the array of the half-lengths of the modified Daniell filters to be applied.

All the values in SMOOTH_PARAM(:) must be greater than 0 and less than (size(VEC)/2)+1 .

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+1 The means of the time series are removed before computing the power and cross spectra
- TREND=+2 The drifts from time series are removed before computing the power and cross spectra
- TREND=+3 The least-squares lines from time series are removed before computing the power and cross spectra.

For other values of TREND nothing is done before estimating the power and cross spectra.

The default is TREND=1, e.g. the means of the time series are removed before the computations.

WIN (INPUT, OPTIONAL) integer(i4b) On entry, this argument specify the data window used in the computations of the power and cross spectra. If:

- WIN=+1 The Bartlett window is used
- WIN=+2 The square window is used
- WIN=+3 The Welch window is used
- WIN=+4 The Hann window is used
- WIN=+5 The Hamming window is used
- WIN=+6 A split-cosine-bell window is used

The default is WIN=3, e.g. the Welch window is used.

TAPERP (INPUT, OPTIONAL) real(stnd) The total percentage of the data to be tapered if WIN=6. TAPERP must be greater than zero and less or equal to one, otherwise the default value is used.

The default is 0.2 .

PROBTEST (INPUT, OPTIONAL) real(stnd) On entry, a probability. PROBTEST is the critical probability which is used to determine the lower and upper confidence limit factors (e.g. the optional

arguments CONLWR and CONUPR) and the critical value for testing the null hypothesis that the squared coherency is zero (e.g. the TESTCOHER optional argument).

PROBTEST must verify: 0. < P < 1.

The default is 0.05 .

Further Details

After removing the mean or the trend from the time series (e.g. TREND=1,2,3), the selected data window (e.g. WIN=1,2,3,4,5,6) is applied to the time series and the PSD and CSD estimates are computed by the FFT of these transformed time series. Optionally, theses PSD and CSD estimates may then be smoothed in the frequency domain by modified Daniell filters (e.g. if argument SMOOTH_PARAM is used).

The computed equivalent number of degrees of freedom and bandwidth must be divided by two for the zero and Nyquist frequencies.

Furthermore, the computed equivalent number of degrees of freedom, bandwidth, lower and upper (1-PROBTEST) * 100% confidence limit factors and critical value for the squared coherency (e.g. arguments EDOF, BANDWIDTH, CONLWR, CONUPR and TESTCOHER) are not right near the zero and Nyquist frequencies if the PSD estimates have been smoothed by modified Daniell filters. The reason is that CROSS_SPECTRUM assumes that smoothing involves averaging independent frequency ordinates. This is true except near the zero and Nyquist frequencies where an average may contain contributions from negative frequencies, which are identical to and hence not independent of positive frequency spectral values. Thus, the number of degrees of freedom in PSD estimates near the 0 and Nyquist frequencies are as little as half the number of degrees of freedom of the spectral estimates away from these frequency extremes if the optional argument SMOOTH_PARAM is used.

If the optional argument SMOOTH_PARAM is used, the computed equivalent number of degrees of freedom, bandwidth, lower and upper (1-PROBTEST) * 100% confidence limit factors and critical value for the squared coherency are right for PSD estimates at frequencies

(i-1)/size(VEC) for i= (nparam+1)/2 + 1 to (size(VEC) - nparam + 1)/2

where nparam = 2 * (2+sum(SMOOTH_PARAM(:)))- 1, (e.g. for frequencies i/size(VEC) for i = (nparam+1)/2, . . . , (size(VEC)-nparam-1)/2) .

For definitions, more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York.

(2) **Welch, P.D., 1967:** The use of Fast Fourier Transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. IEEE trans. on audio and electroacoustics, Vol. Au-15, 2, 70-73.

(3) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford.

6.27.67 subroutine power_spectrum2 (vec, l, psvec, freq, edof, bandwidth, conlwr, conupr, initfft, overlap, normpsd, smooth_param, trend, trend2, win, taperp, l0, probtest)

Purpose

Subroutine POWER_SPECTRUM2 computes a Fast Fourier Transform (FFT) estimate of the power spectrum of a real time series.

The Power Spectral Density (PSD) estimates are returned in units which are the square of the data (if NORMPSD=false) or in spectral density units (if NORMPSD=true).

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the real time series for which the power spectrum must be estimated. If TREND=1, 2 or 3, VEC is used as workspace and is transformed.

Size(VEC) must be greater or equal to 4.

L (INPUT) integer(i4b) On entry, an integer used to segment the time series. L is the length of the segments. L must be a positive even integer, less or equal to size(VEC), but greater or equal to 4.

Spectral computations are at (L/2)+1 frequencies if the optional argument L0 is absent and are at ((L+L0)/2)+1 frequencies if L0 is present (L0 is the number of zeros added to each segment).

Suggested values for L+L0 are 16, 32, 64 or 128 (e.g. an integer power of two, in order to speed the computations).

PSVEC (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the Power Spectral Density (PSD) estimates of VEC.

PSVEC must verify: size(PSVEC) = ((L+L0)/2) + 1 .

FREQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the frequencies at which the spectral quantities are calculated in cycles per unit of time.

The spectral estimates are taken at frequencies (i-1)/(L+L0) for i=1,2, . . . , ((L+L0)/2 + 1).

FREQ must verify: size(FREQ) = (L+L0)/2 + 1 .

EDOF (OUTPUT, OPTIONAL) real(stnd) On exit, the equivalent number of degrees of freedom of the power spectrum estimates.

BANDWIDTH (OUTPUT, OPTIONAL) real(stnd) On exit, the bandwidth of the power spectrum estimates.

CONLWR (OUTPUT, OPTIONAL) real(stnd)

CONUPR (OUTPUT, OPTIONAL) real(stnd) On output, these arguments specify the lower and upper (1-PROBTEST) * 100% confidence limit factors, respectively. Multiply the PSD estimates (e.g. the PSVEC(:) argument) by these constants to get the lower and upper limits of a (1-PROBTEST) * 100% confidence interval for the PSD estimates.

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITFFT = false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine POWER_SPECTRUM2 in order to sets up constants and functions for use by subroutine FFT which is called inside subroutine POWER_SPECTRUM2. This call to INITFFT must have the following form:

  call init_fft((L+L0)/2)

- INITFFT = true, the call to INIT_FFT is done inside subroutine POWER_SPECTRUM2 and a call to END_FFT is also done before leaving subroutine POWER_SPECTRUM2.

The default is INITFFT=true .

OVERLAP (INPUT, OPTIONAL) logical(lgl) If:

- OVERLAP = false, the subroutine segments the data without any overlapping.
- OVERLAP = true, the subroutine overlaps the segments by one half of their length (which is equal to L).

In both cases, zeros are eventually added to each segment (if argument L0 is present) and each segment will be FFT'd, and the resulting periodograms will averaged together to obtain a Power Spectrum Density estimate at the ((L+L0)/2)+1 frequencies.

The default is OVERLAP=false .

NORMPSD (INPUT, OPTIONAL) logical(lgl) On entry, if:

- NORMPSD = true, the PSD estimates are normalized in such a way that the total area under the power spectrum is equal to the variance of the time series VEC.
- NORMPSD = false, the sum of the PSD estimates (e.g. sum(PSVEC(2:)) is equal to the variance of the time series.

The default is NORMPSD=true .

SMOOTH_PARAM (INPUT, OPTIONAL) integer(i4b), dimension(:) If SMOOTH_PARAM is used, the PSD estimates are computed by repeated smoothing of the periodogram with modified Daniell weights.

On entry, SMOOTH_PARAM(:) gives the array of the half-lengths of the modified Daniell filters to be applied.

All the values in SMOOTH_PARAM(:) must be greater than 0 and less than ((L+L0)/2) + 1 .

Size(SMOOTH_PARAM) must be greater or equal to 1.

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+1 The mean of the time series is removed before computing the spectrum
- TREND=+2 The drift from the time series is removed before computing the spectrum by using the formula: drift = (VEC(size(VEC)) - VEC(1))/(size(VEC) - 1)
- TREND=+3 The least-squares line from the time series is removed before computing the spectrum.

For other values of TREND nothing is done before estimating the power spectrum.

The default is TREND=1, e.g. the mean of the time series is removed before the computations.

TREND2 (INPUT, OPTIONAL) integer(i4b) If:

- TREND2=+1 The mean of the time segment is removed before computing the spectrum on this segment.
- TREND2=+2 The drift from the time segment is removed before computing the spectrum on this segment.
- TREND2=+3 The least-squares line from the time segment is removed before computing the spectrum on this segment.

For other values of TREND2 nothing is done before estimating the power spectrum on each segment.

The default is TREND2=0, e.g. nothing is done before estimating the power spectrum on each segment.

WIN (INPUT, OPTIONAL) integer(i4b) On entry, this argument specify the data window used in the computations of the power spectrum. If:

- WIN=+1 The Bartlett window is used
- WIN=+2 The square window is used
- WIN=+3 The Welch window is used
- WIN=+4 The Hann window is used

- WIN=+5 The Hamming window is used
- WIN=+6 A split-cosine-bell window is used

The default is WIN=3, e.g. the Welch window is used.

TAPERP (INPUT, OPTIONAL) real(stnd) The total percentage of the data to be tapered if WIN=6. TAPERP must be greater than zero and less or equal to one, otherwise the default value is used.

The default is 0.2 .

L0 (INPUT, OPTIONAL) integer(i4b) The number of zeros added to each time segment in order to obtain more finely spaced spectral estimates. L+L0 must be a positive even integer.

The default is L0=0, e.g. no zeros are added to each time segment.

PROBTEST (INPUT, OPTIONAL) real(stnd) On entry, a probability. PROBTEST is the critical probability which is used to determine the lower and upper confidence limit factors (e.g. the optional arguments CONLWR and CONUPR).

PROBTEST must verify: 0. < P < 1.

The default is 0.05 .

Further Details

After removing the mean or the trend from the time series (e.g. TREND=1,2,3), the series is padded with zero on the right such that the length of the resulting time series is evenly divisible by L (a positive even integer). The length, N, of this resulting time series is the first integer greater than or equal to size(VEC) which is evenly divisible by L. If size(VEC) is not evenly divisible by L, N is equal to size(VEC)+L-mod(size(VEC),L).

Optionally, the mean or the trend may also be removed from each time segment (e.g. TREND2=1,2,3). Optionally, zeros may be added to each time segment (e.g. the optional arguemnt L0) if more finely spaced spectral esimates are desired.

The stability of the PSD estimates depends on the averaging process. That is, the greater the number of segments (N/L if OVERLAP=false and (2N/L)-1 if OVERLAP=true), the more stable the resulting PSD estimates.

Optionally, theses PSD estimates may then be smoothed again in the frequency domain by modified Daniell filters (e.g. if argument SMOOTH_PARAM is used).

The computed equivalent number of degrees of freedom and bandwidth must be divided by two for the zero and Nyquist frequencies.

Furthermore, the computed equivalent number of degrees of freedom, bandwidth, lower and upper (1-PROBTEST) * 100% confidence limit factors are not right near the zero and Nyquist frequencies if the PSD estimates have been smoothed by modified Daniell filters. The reason is that POWER_SPECTRUM2 assumes that smoothing involves averaging independent frequency ordinates. This is true except near the zero and Nyquist frequencies where an average may contain contributions from negative frequencies, which are identical to and hence not independent of positive frequency spectral values. Thus, the number of degrees of freedom in PSD estimates near the 0 and Nyquist frequencies are as little as half the number of degrees of freedom of the spectral estimates away from these frequency extremes if the optional argument SMOOTH_PARAM is used.

If the optional argument SMOOTH_PARAM is used, the computed equivalent number of degrees of freedom, bandwidth, lower and upper (1-PROBTEST) * 100% confidence limit factors are right for PSD estimates at frequencies

(i-1)/(L+L0) for i= (nparam+1)/2 + 1 to ((L+L0) - nparam + 1)/2

where nparam = 2 * (2+sum(SMOOTH_PARAM(:)))- 1, (e.g. for frequencies i/(L+L0) for i = (nparam+1)/2, . . . , ((L+L0) - nparam - 1)/2).

For definitions, more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York.

(2) **Welch, P.D., 1967:** The use of Fast Fourier Transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. IEEE trans. on audio and electroacoustics, Vol. Au-15, 2, 70-73.

(3) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford.

6.27.68 subroutine power_spectrum2 (mat, l, psmat, freq, edof, bandwidth, conlwr, conupr, initfft, overlap, normpsd, smooth_param, trend, trend2, win, taperp, l0, probtest)

Purpose

Subroutine POWER_SPECTRUM2 computes Fast Fourier Transform (FFT) estimates of the power spectra of the multi-channel real time series MAT (e.g. each row of MAT contains a time series).

The Power Spectral Density (PSD) estimates are returned in units which are the square of the data (if NORMPSD=false) or in spectral density units (if NORMPSD=true).

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the multi-channel real time series for which the power spectra must be estimated. Each row of MAT is a real time series. If TREND=1, 2 or 3, MAT is used as workspace and is transformed.

Size(MAT,2) must be greater or equal to 4.

L (INPUT) integer(i4b) On entry, an integer used to segment the time series. L is the length of the segments. L must be a positive even integer less or equal to size(MAT,2), but greater or equal to 4.

Spectral computations are at (L/2)+1 frequencies if the optional argument L0 is absent and are at ((L+L0)/2)+1 frequencies if L0 is present (L0 is the number of zeros added to each segment).

Suggested values for L+L0 are 16, 32, 64 or 128 (e.g. an integer power of two, in order to speed the computations).

PSMAT (OUTPUT) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and ((L+L0)/2) + 1 columns containing the Power Spectral Density (PSD) estimates of each row of MAT.

The shape of PSMAT must verify:

- size(PSMAT,1) = size(MAT,1) ;
- size(PSMAT,2) = ((L+L0)/2) + 1 .

FREQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the frequencies at which the spectral quantities are calculated in cycles per unit of time.

The spectral estimates are taken at frequencies (i-1)/(L+L0) for i=1,2, . . . , ((L+L0)/2 + 1).

FREQ must verify: size(FREQ) = (L+L0)/2 + 1 .

EDOF (OUTPUT, OPTIONAL) real(stnd) On exit, the equivalent number of degrees of freedom of the power spectrum estimates.

BANDWIDTH (OUTPUT, OPTIONAL) real(stnd) On exit, the bandwidth of the power spectrum estimates.

CONLWR (OUTPUT, OPTIONAL) real(stnd)

CONUPR (OUTPUT, OPTIONAL) real(stnd) On output, these arguments specify the lower and upper (1-PROBTEST) * 100% confidence limit factors, respectively. Multiply the PSD estimates (e.g. the PSMAT(:,:) argument) by these constants to get the lower and upper limits of a (1-PROBTEST) * 100% confidence interval for the PSD estimates.

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITFFT = false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine POWER_SPECTRUM2 in order to sets up constants and functions for use by subroutine FFT which is called inside subroutine POWER_SPECTRUM2. This call to INITFFT must have the following form:

  call init_fft((/ size(MAT,1), (L+L0)/2 /), dim=2_i4b)

- INITFFT = true, the call to INIT_FFT is done inside subroutine POWER_SPECTRUM2 and a call to END_FFT is also done before leaving subroutine POWER_SPECTRUM2.

The default is INITFFT=true .

OVERLAP (INPUT, OPTIONAL) logical(lgl) If:

- OVERLAP = false, the subroutine segments the data without any overlapping.
- OVERLAP = true, the subroutine overlaps the segments by one half of their length (which is equal to L).

In both cases, zeros are eventually added to each segment (if argument L0 is present) and each segment will be FFT'd, and the resulting periodograms will averaged together to obtain a Power Spectrum Density estimate at the ((L+L0)/2)+1 frequencies.

The default is OVERLAP=false .

NORMPSD (INPUT, OPTIONAL) logical(lgl) On entry, if:

- NORMPSD = true, the PSD estimates are normalized in such a way that the total area under the power spectrum is equal to the variance of the corresponding time series in MAT.
- NORMPSD = false, the sum of the PSD estimates (e.g. sum(PSMAT(:,2:),dim=2)) is equal to the variance of the corresponding time series.

The default is NORMPSD=true .

SMOOTH_PARAM (INPUT, OPTIONAL) integer(i4b), dimension(:) If SMOOTH_PARAM is used, the PSD estimates are computed by repeated smoothing of the periodogram with modified Daniell weights.

On entry, SMOOTH_PARAM(:) gives the array of the half-lengths of the modified Daniell filters to be applied.

All the values in SMOOTH_PARAM(:) must be greater than 0 and less than ((L+L0)/2)+1 .

Size(SMOOTH_PARAM) must be greater or equal to 1.

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+1 The means of the time series are removed before computing the spectra
- TREND=+2 The drifts from time series are removed before computing the spectra

- TREND=+3 The least-squares lines from time series are removed before computing the spectra.

For other values of TREND nothing is done before estimating the power and cross spectra.

The default is TREND=1, e.g. the means of the time series are removed before the computations.

TREND2 (INPUT, OPTIONAL) integer(i4b) If:

- TREND2=+1 The mean of the time segment is removed before computing the spectrum on this segment.
- TREND2=+2 The drift from the time segment is removed before computing the spectrum on this segment.
- TREND2=+3 The least-squares line from the time segment is removed before computing the spectrum on this segment.

For other values of TREND2 nothing is done before estimating the power spectrum on each segment.

The default is TREND2=0, e.g. nothing is done before estimating the power spectrum on each segment.

WIN (INPUT, OPTIONAL) integer(i4b) On entry, this argument specify the data window used in the computations of the power spectrum. If:

- WIN=+1 The Bartlett window is used
- WIN=+2 The square window is used
- WIN=+3 The Welch window is used
- WIN=+4 The Hann window is used
- WIN=+5 The Hamming window is used
- WIN=+6 A split-cosine-bell window is used

The default is WIN=3, e.g. the Welch window is used.

TAPERP (INPUT, OPTIONAL) real(stnd) The total percentage of the data to be tapered if WIN=6. TAPERP must be greater than zero and less or equal to one, otherwise the default value is used.

The default is 0.2 .

L0 (INPUT, OPTIONAL) integer(i4b) The number of zeros added to each time segment in order to obtain more finely spaced spectral estimates. L+L0 must be a positive even integer.

The default is L0=0, e.g. no zeros are added to each time segment.

PROBTEST (INPUT, OPTIONAL) real(stnd) On entry, a probability. PROBTEST is the critical probability which is used to determine the lower and upper confidence limit factors (e.g. the optional arguments CONLWR and CONUPR).

PROBTEST must verify: 0. < P < 1.

The default is 0.05 .

Further Details

After removing the mean or the trend from the time series (e.g. TREND=1,2,3), the series are padded with zero on the right such that the length of the resulting time series is evenly divisible by L (a positive even integer). The length, N, of this resulting time series is the first integer greater than or equal to size(MAT,2) which is evenly divisible by L. If size(MAT,2) is not evenly divisible by L, N is equal to size(MAT,2)+L-mod(size(MAT,2),L).

Optionally, the mean or the trend may also be removed from each time segment (e.g. TREND2=1,2,3). Optionally, zeros may be added to each time segment (e.g. the optional arguemnt L0) if more finely spaced spectral esimates are desired.

The stability of the PSD estimates depends on the averaging process. That is, the greater the number of segments (N/L if OVERLAP=false and (2N/L)-1 if OVERLAP=true), the more stable the resulting PSD estimates.

Optionally, theses PSD estimates may then be smoothed again in the frequency domain by modified Daniell filters (e.g. if argument SMOOTH_PARAM is used).

The computed equivalent number of degrees of freedom and bandwidth must be divided by two for the zero and Nyquist frequencies.

Furthermore, the computed equivalent number of degrees of freedom, bandwidth, lower and upper (1-PROBTEST) * 100% confidence limit factors are not right near the zero and Nyquist frequencies if the PSD estimates have been smoothed by modified Daniell filters. The reason is that POWER_SPECTRUM2 assumes that smoothing involves averaging independent frequency ordinates. This is true except near the zero and Nyquist frequencies where an average may contain contributions from negative frequencies, which are identical to and hence not independent of positive frequency spectral values. Thus, the number of degrees of freedom in PSD estimates near the 0 and Nyquist frequencies are as little as half the number of degrees of freedom of the spectral estimates away from these frequency extremes if the optional argument SMOOTH_PARAM is used.

If the optional argument SMOOTH_PARAM is used, the computed equivalent number of degrees of freedom, bandwidth, lower and upper (1-PROBTEST) * 100% confidence limit factors are right for PSD estimates at frequencies

(i-1)/(L+L0) for i= (nparam+1)/2 + 1 to ((L+L0) - nparam + 1)/2

where nparam = 2 * (2+sum(SMOOTH_PARAM(:)))- 1, (e.g. for frequencies i/(L+L0) for i = (nparam+1)/2, . . . , ((L+L0) - nparam - 1)/2).

For definitions, more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York.

(2) **Welch, P.D., 1967:** The use of Fast Fourier Transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. IEEE trans. on audio and electroacoustics, Vol. Au-15, 2, 70-73.

(3) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford.

6.27.69 subroutine cross_spectrum2 (vec, vec2, l, psvec, psvec2, phase, coher, freq, edof, bandwidth, conlwr, conupr, testcoher, ampli, co_spect, quad_spect, prob_coher, initfft, overlap, normpsd, smooth_param, trend, trend2, win, taperp, l0, probtest)

Purpose

Subroutine CROSS_SPECTRUM2 computes Fast Fourier Transform (FFT) estimates of the power and cross spectra of two real time series.

The Power Spectral Density (PSD) and Cross Spectral Density (CSD) estimates are returned in units which are the square of the data (if NORMPSD=false) or in spectral density units (if NORMPSD=true).

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the first real time series for which the power and cross spectra must be estimated. If TREND=1, 2 or 3, VEC is used as workspace and is transformed.

Size(VEC) must be greater or equal to 4.

VEC2 (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the second real time series for which the power and cross spectra must be estimated. If TREND=1, 2 or 3, VEC2 is used as workspace and is transformed.

VEC2 must verify: size(VEC2) = size(VEC).

L (INPUT) integer(i4b) On entry, an integer used to segment the time series. L is the length of the segments. L must be a positive even integer, less or equal to size(VEC), but greater or equal to 4.

Spectral computations are at (L/2)+1 frequencies if the optional argument L0 is absent and are at ((L+L0)/2)+1 frequencies if L0 is present (L0 is the number of zeros added to each segment).

Suggested values for L+L0 are 16, 32, 64 or 128 (e.g. an integer power of two, in order to speed the computations).

PSVEC (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the Power Spectral Density (PSD) estimates of VEC.

PSVEC must verify: size(PSVEC) = ((L+L0)/2) + 1 .

PSVEC2 (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the Power Spectral Density (PSD) estimates of VEC2.

PSVEC2 must verify: size(PSVEC2) = ((L+L0)/2) + 1 .

PHASE (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the phase of the cross spectrum, given in fractions of a circle (e.g. on the closed interval (0,1)).

PHASE must verify: size(PHASE) = ((L+L0)/2) + 1 .

COHER (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the squared coherency estimates for all frequencies.

COHER must verify: size(COHER) = ((L+L0)/2) + 1 .

FREQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the frequencies at which the spectral quantities are calculated in cycles per unit of time.

The spectral estimates are taken at frequencies (i-1)/(L+L0) for i=1,2, . . . , ((L+L0)/2 + 1).

FREQ must verify: size(FREQ) = (L+L0)/2 + 1 .

EDOF (OUTPUT, OPTIONAL) real(stnd) On exit, the equivalent number of degrees of freedom of the power and cross spectrum estimates.

BANDWIDTH (OUTPUT, OPTIONAL) real(stnd) On exit, the bandwidth of the power and cross spectrum estimates.

CONLWR (OUTPUT, OPTIONAL) real(stnd)

CONUPR (OUTPUT, OPTIONAL) real(stnd) On output, these arguments specify the lower and upper (1-PROBTEST) * 100% confidence limit factors, respectively. Multiply the PSD estimates (e.g. the PSVEC(:) and PSVEC2(:) arguments) by these constants to get the lower and upper limits of a (1-PROBTEST) * 100% confidence interval for the PSD estimates.

TESTCOHER (OUTPUT, OPTIONAL) real(stnd) On output, this argument specifies the critical value for testing the null hypothesis that the squared coherency is zero at the PROBTEST * 100% significance level (e.g. elements of COHER(:) less than TESTCOHER should be regarded as not significantly different from zero at the PROBTEST * 100% significance level).

AMPLI (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the cross-amplitude spectrum.

AMPLI must verify: size(AMPLI) = ((L+L0)/2) + 1 .

CO_SPECT (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the co-spectrum (e.g. the real part of cross-spectrum).

CO_SPECT must verify: size(CO_SPECT) = ((L+L0)/2) + 1 .

QUAD_SPECT (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the quadrature spectrum (e.g. the imaginary part of cross-spectrum with a minus sign).

QUAD_SPECT must verify: size(QUAD_SPECT) = ((L+L0)/2) + 1 .

PROB_COHER (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the probabilities that the computed sample squared coherencies came from an ergodic stationary bivariate process with (corresponding) squared coherencies equal to zero.

PROB_COHER must verify: size(PROB_COHER) = ((L+L0)/2)+1 .

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if INITFFT is set to false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine CROSS_SPECTRUM2 in order to sets up constants and functions for use by subroutine FFT which is called inside subroutine CROSS_SPECTRUM2 (the call to INITFFT must have the following form:

```
call init_fft( (L+L0)/2 )
```

If INITFFT is set to true, the call to INIT_FFT is done inside subroutine CROSS_SPECTRUM2 and a call to END_FFT is also done before leaving subroutine CROSS_SPECTRUM2.

The default is INITFFT=true .

OVERLAP (INPUT, OPTIONAL) logical(lgl) If:

- OVERLAP = false, the subroutine segments the data without any overlapping.
- OVERLAP = true, the subroutine overlaps the segments by one half of their length (which is equal to L).

In both cases, zeros are eventually added to each segment (if argument L0 is present) and each segment will be FFT'd, and the resulting periodograms will averaged together to obtain a Power Spectrum Density estimate at the ((L+L0)/2)+1 frequencies.

The default is OVERLAP=false .

NORMPSD (INPUT, OPTIONAL) logical(lgl) On entry, if NORMPSD is set to true, the power and cross spectra estimates are normalized in such a way that the total area under the power spectrum is equal to the variance of the time series VEC and VEC2. If NORMPSD is set to false, the sum of the PSD estimates (e.g. sum(PSVEC(2:)) and sum(PSVEC2(2:))) is equal to the variance of the corresponding time series.

The default is NORMPSD=true .

SMOOTH_PARAM (INPUT, OPTIONAL) integer(i4b), dimension(:) If SMOOTH_PARAM is used, the power and cross spectra estimates are computed by repeated smoothing of the periodograms and cross-periodogram with modified Daniell weights.

On entry, SMOOTH_PARAM(:) gives the array of the half-lengths of the modified Daniell filters to be applied.

All the values in SMOOTH_PARAM(:) must be greater than 0 and less than (L+L0)/2 + 1 .

Size(SMOOTH_PARAM) must be greater or equal to 1.

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+1 The mean of the two time series is removed before computing the spectra
- TREND=+2 The drift from the two time series is removed before computing the spectra
- TREND=+3 The least-squares line from the two time series is removed before computing the spectra.

For other values of TREND nothing is done before estimating the power and cross spectra.

The default is TREND=1, e.g. the means of the time series are removed before the computations.

TREND2 (INPUT, OPTIONAL) integer(i4b) If:

- TREND2=+1 The mean of the time segment is removed before computing the cross-spectrum on this segment.
- TREND2=+2 The drift from the time segment is removed before computing the cross-spectrum on this segment.
- TREND2=+3 The least-squares line from the time segment is removed before computing the cross-spectrum on this segment.

For other values of TREND2 nothing is done before estimating the cross-spectrum on each segment.

The default is TREND2=0, e.g. nothing is done before estimating the power spectrum on each segment.

WIN (INPUT, OPTIONAL) integer(i4b) On entry, this argument specify the data window used in the computations of the power and cross spectra. If:

- WIN=+1 The Bartlett window is used
- WIN=+2 The square window is used
- WIN=+3 The Welch window is used
- WIN=+4 The Hann window is used
- WIN=+5 The Hamming window is used
- WIN=+6 A split-cosine-bell window is used

The default is WIN=3, e.g. the Welch window is used.

TAPERP (INPUT, OPTIONAL) real(stnd) The total percentage of the data to be tapered if WIN=6. TAPERP must be greater than zero and less or equal to one, otherwise the default value is used.

The default is 0.2 .

L0 (INPUT, OPTIONAL) integer(i4b) The number of zeros added to each time segment in order to obtain more finely spaced spectral estimates. L+L0 must be a positive even integer.

The default is L0=0, e.g. no zeros are added to each time segment.

PROBTEST (INPUT, OPTIONAL) real(stnd) On entry, a probability. PROBTEST is the critical probability which is used to determine the lower and upper confidence limit factors (e.g. the optional arguments CONLWR and CONUPR) and the critical value for testing the null hypothesis that the squared coherency is zero (e.g. the TESTCOHER optional argument).

PROBTEST must verify: 0. < P < 1.

The default is 0.05 .

Further Details

After removing the mean or the trend from the two time series (e.g. TREND=1,2,3), the series are padded with zero on the right such that the length of the resulting two time series is evenly divisible by L (a positive even integer). The length, N, of these resulting time series is the first integer greater than or equal to size(VEC) which is evenly divisible by L. If size(VEC) is not evenly divisible by L, N is equal to size(VEC)+L-mod(size(VEC),L).

Optionally, the mean or the trend may also be removed from each time segment (e.g. TREND2=1,2,3). Optionally, zeros may be added to each time segment (e.g. the optional arguemnt L0) if more finely spaced spectral esimates are desired.

The stability of the power and cross spectra estimates depends on the averaging process. That is, the greater the number of segments (N/L if OVERLAP=false and (2N/L)-1 if OVERLAP=true), the more stable the resulting power and cross spectra estimates.

Optionally, these power and cross spectra estimates may then be smoothed again in the frequency domain by modified Daniell filters (e.g. if argument SMOOTH_PARAM is used).

The computed equivalent number of degrees of freedom and bandwidth must be divided by two for the zero and Nyquist frequencies.

Furthermore, the computed equivalent number of degrees of freedom, bandwidth, lower and upper (1-PROBTEST) * 100% confidence limit factors and critical value for the squared coherency (e.g. arguments EDOF, BANDWIDTH, CONLWR, CONUPR and TESTCOHER) are not right near the zero and Nyquist frequencies if the PSD estimates have been smoothed by modified Daniell filters. The reason is that CROSS_SPECTRUM2 assumes that smoothing involves averaging independent frequency ordinates. This is true except near the zero and Nyquist frequencies where an average may contain contributions from negative frequencies, which are identical to and hence not independent of positive frequency spectral values. Thus, the number of degrees of freedom in PSD estimates near the 0 and Nyquist frequencies are as little as half the number of degrees of freedom of the spectral estimates away from these frequency extremes if the optional argument SMOOTH_PARAM is used.

If the optional argument SMOOTH_PARAM is used, the computed equivalent number of degrees of freedom, bandwidth, lower and upper (1-PROBTEST) * 100% confidence limit factors and critical value for the squared coherency are right for PSD estimates at frequencies

(i-1)/(L+L0) for i= (nparam+1)/2 + 1 to ((L+L0) - nparam + 1)/2

where nparam = 2 * (2+sum(SMOOTH_PARAM(:)))- 1, (e.g. for frequencies i/(L+L0) for i = (nparam+1)/2, . . . , ((L+L0)-nparam-1)/2) .

For definitions, more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York.

(2) **Welch, P.D., 1967:** The use of Fast Fourier Transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. IEEE trans. on audio and electroacoustics, Vol. Au-15, 2, 70-73.

(3) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford.

6.27.70 subroutine cross_spectrum2 (vec, mat, l, psvec, psmat, phase, coher, freq, edof, bandwidth, conlwr, conupr, testcoher, ampli, co_spect, quad_spect, prob_coher, initfft, overlap, normpsd, smooth_param, trend, trend2, win, taperp, l0, probtest)

Purpose

Subroutine CROSS_SPECTRUM2 computes Fast Fourier Transform (FFT) estimates of the power and cross spectra of the real time series, VEC, and the multi-channel real time series MAT.

The Power Spectral Density (PSD) and Cross Spectral Density (CSD) estimates are returned in units which are the square of the data (if NORMPSD=false) or in spectral density units (if NORMPSD=true).

Arguments

VEC (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the real time series for which the power and cross spectra must be estimated. If TREND=1, 2 or 3, VEC is used as workspace and is transformed.

Size(VEC) must be greater or equal to 4.

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the multi-channel real time series for which the power and cross spectra must be estimated. Each row of MAT is a real time series. If TREND=1, 2 or 3, MAT is used as workspace and is transformed.

The shape of MAT must verify: size(MAT,2) = size(VEC).

L (INPUT) integer(i4b) On entry, an integer used to segment the time series. L is the length of the segments. L must be a positive even integer, less or equal to size(VEC), but greater or equal to 4.

Spectral computations are at (L/2)+1 frequencies if the optional argument L0 is absent and are at ((L+L0)/2)+1 frequencies if L0 is present (L0 is the number of zeros added to each segment).

Suggested values for L+L0 are 16, 32, 64 or 128 (e.g. an integer power of two, in order to speed the computations).

PSVEC (OUTPUT) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the Power Spectral Density (PSD) estimates of VEC.

PSVEC must verify: size(PSVEC) = ((L+L0)/2) + 1 .

PSMAT (OUTPUT) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and ((L+L0)/2) + 1 columns containing the Power Spectral Density (PSD) estimates of each row of MAT.

The shape of PSMAT must verify:

- size(PSMAT,1) = size(MAT,1) ;
- size(PSMAT,2) = ((L+L0)/2) + 1 .

PHASE (OUTPUT) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and ((L+L0)/2) + 1 columns containing the phase of the cross spectrum, given in fractions of a circle (e.g. on the closed interval (0,1)).

The shape of PHASE must verify:

- size(PHASE,1) = size(MAT,1) ;
- size(PHASE,2) = ((L+L0)/2) + 1 .

COHER (OUTPUT) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and ((L+L0)/2) + 1 columns containing the squared coherency estimates for all frequencies.

The shape of COHER must verify:

- size(COHER,1) = size(MAT,1) ;
- size(COHER,2) = ((L+L0)/2) + 1 .

FREQ (OUTPUT, OPTIONAL) real(stnd), dimension(:) On exit, a real vector of length ((L+L0)/2)+1 containing the frequencies at which the spectral quantities are calculated in cycles per unit of time.

The spectral estimates are taken at frequencies (i-1)/(L+L0) for i=1,2, . . . , ((L+L0)/2 + 1).

FREQ must verify: size(FREQ) = (L+L0)/2 + 1 .

EDOF (OUTPUT, OPTIONAL) real(stnd) On exit, the equivalent number of degrees of freedom of the power and cross spectrum estimates.

BANDWIDTH (OUTPUT, OPTIONAL) real(stnd) On exit, the bandwidth of the power and cross spectrum estimates.

CONLWR (OUTPUT, OPTIONAL) real(stnd)

CONUPR (OUTPUT, OPTIONAL) real(stnd) On output, these arguments specify the lower and upper (1-PROBTEST) * 100% confidence limit factors, respectively. Multiply the PSD estimates (e.g. the PSVEC(:) and PSMAT(:,:) arguments) by these constants to get the lower and upper limits of a (1-PROBTEST) * 100% confidence interval for the PSD estimates.

TESTCOHER (OUTPUT, OPTIONAL) real(stnd) On output, this argument specifies the critical value for testing the null hypothesis that the squared coherency is zero at the PROBTEST * 100% significance level (e.g. elements of COHER(:,:) less than TESTCOHER should be regarded as not significantly different from zero at the PROBTEST * 100% significance level).

AMPLI (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and ((L+L0)/2) + 1 columns containing the cross-amplitude spectra.

The shape of AMPLI must verify:

- size(AMPLI,1) = size(MAT,1) ;
- size(AMPLI,2) = ((L+L0)/2) + 1 .

CO_SPECT (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and ((L+L0)/2) + 1 columns containing the co-spectra (e.g. the real part of cross-spectra).

The shape of CO_SPECT must verify:

- size(CO_SPECT,1) = size(MAT,1) ;
- size(CO_SPECT,2) = ((L+L0)/2) + 1 .

QUAD_SPECT (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and ((L+L0)/2) + 1 columns containing the quadrature spectrum (e.g. the imaginary part of cross-spectrum with a minus sign).

The shape of QUAD_SPECT must verify:

- size(QUAD_SPECT,1) = size(MAT,1) ;
- size(QUAD_SPECT,2) = ((L+L0)/2) + 1 .

PROB_COHER (OUTPUT, OPTIONAL) real(stnd), dimension(:,:) On exit, a real matrix with size(MAT,1) rows and ((L+L0)/2) + 1 columns containing the probabilities that the computed sample

squared coherencies came from an ergodic stationary bivariate process with (corresponding) squared coherencies equal to zero.

The shape of PROB_COHER must verify:

- size(PROB_COHER,1) = size(MAT,1) ;
- size(PROB_COHER,2) = ((L+L0)/2) + 1 .

INITFFT (INPUT, OPTIONAL) logical(lgl) On entry, if:

- INITFFT = false, it is assumed that a call to subroutine INIT_FFT has been done before calling subroutine CROSS_SPECTRUM2 in order to sets up constants and functions for use by subroutine FFT which is called inside subroutine CROSS_SPECTRUM2. This call to INITFFT must have the following form:

  call init_fft((/ size(MAT,1), (L+L0)/2 /), dim=2_i4b)

- INITFFT = true, the call to INIT_FFT is done inside subroutine CROSS_SPECTRUM2 and a call to END_FFT is also done before leaving subroutine CROSS_SPECTRUM2.

The default is INITFFT=true .

OVERLAP (INPUT, OPTIONAL) logical(lgl) If:

- OVERLAP = false, the subroutine segments the data without any overlapping.
- OVERLAP = true, the subroutine overlaps the segments by one half of their length (which is equal to L).

In both cases, zeros are eventually added to each segment (if argument L0 is present) and each segment will be FFT'd, and the resulting periodograms will averaged together to obtain a Power Spectrum Density estimate at the ((L+L0)/2)+1 frequencies.

The default is OVERLAP=false .

NORMPSD (INPUT, OPTIONAL) logical(lgl) On entry, if:

- NORMPSD = true, the power and cross spectra estimates are normalized in such a way that the total area under the power spectra is equal to the variance of the time series contained in VEC and in each row of MAT.
- NORMPSD = false, the sum of the PSD estimates (e.g. sum(PSVEC(2:)) and sum(PSMAT(:,2:),dim=2)) is equal to the variance of the corresponding time series.

The default is NORMPSD=true .

SMOOTH_PARAM (INPUT, OPTIONAL) integer(i4b), dimension(:) If SMOOTH_PARAM is used, the power and cross spectra estimates are computed by repeated smoothing of the periodograms and cross-periodogram with modified Daniell weights.

On entry, SMOOTH_PARAM(:) gives the array of the half-lengths of the modified Daniell filters to be applied.

All the values in SMOOTH_PARAM(:) must be greater than 0 and less than ((L+L0)/2)+1 .

TREND (INPUT, OPTIONAL) integer(i4b) If:

- TREND=+1 The means of the time series are removed before computing the spectra
- TREND=+2 The drifts from time series are removed before computing the spectra
- TREND=+3 The least-squares lines from time series are removed before computing the spectra.

For other values of TREND nothing is done before estimating the power and cross spectra.

The default is TREND=1, e.g. the means of the time series are removed before the computations.

TREND2 (INPUT, OPTIONAL) integer(i4b) If:

- TREND2=+1 The mean of the time segment is removed before computing the cross-spectrum on this segment.
- TREND2=+2 The drift from the time segment is removed before computing the cross-spectrum on this segment.
- TREND2=+3 The least-squares line from the time segment is removed before computing the cross-spectrum on this segment.

For other values of TREND2 nothing is done before estimating the cross-spectrum on each segment.

The default is TREND2=0, e.g. nothing is done before estimating the power spectrum on each segment.

WIN (INPUT, OPTIONAL) integer(i4b) On entry, this argument specify the data window used in the computations of the power and cross spectra. If:

- WIN=+1 The Bartlett window is used
- WIN=+2 The square window is used
- WIN=+3 The Welch window is used
- WIN=+4 The Hann window is used
- WIN=+5 The Hamming window is used
- WIN=+6 A split-cosine-bell window is used

The default is WIN=3, e.g. the Welch window is used.

TAPERP (INPUT, OPTIONAL) real(stnd) The total percentage of the data to be tapered if WIN=6. TAPERP must be greater than zero and less or equal to one, otherwise the default value is used.

The default is 0.2 .

L0 (INPUT, OPTIONAL) integer(i4b) The number of zeros added to each time segment in order to obtain more finely spaced spectral estimates. L+L0 must be a positive even integer.

The default is L0=0, e.g. no zeros are added to each time segment.

PROBTEST (INPUT, OPTIONAL) real(stnd) On entry, a probability. PROBTEST is the critical probability which is used to determine the lower and upper confidence limit factors (e.g. the optional arguments CONLWR and CONUPR) and the critical value for testing the null hypothesis that the squared coherency is zero (e.g. the TESTCOHER optional argument).

PROBTEST must verify: 0. < P < 1.

The default is 0.05 .

Further Details

After removing the mean or the trend from the time series (e.g. TREND=1,2,3), the series are padded with zero on the right such that the length of the resulting time series is evenly divisible by L (a positive even integer). The length, N, of these resulting time series is the first integer greater than or equal to size(VEC) which is evenly divisible by L. If size(VEC) is not evenly divisible by L, N is equal to size(VEC)+L-mod(size(VEC),L).

Optionally, the mean or the trend may also be removed from each time segment (e.g. TREND2=1,2,3). Optionally, zeros may be added to each time segment (e.g. the optional arguemnt L0) if more finely spaced spectral esimates are desired.

The stability of the power and cross spectra estimates depends on the averaging process. That is, the greater the number of segments (N/L if OVERLAP=false and (2N/L)-1 if OVERLAP=true), the more stable the resulting power and cross spectra estimates.

Optionally, these power and cross spectra estimates may then be smoothed again in the frequency domain by modified Daniell filters (e.g. if argument SMOOTH_PARAM is used).

The computed equivalent number of degrees of freedom and bandwidth must be divided by two for the zero and Nyquist frequencies.

Furthermore, the computed equivalent number of degrees of freedom, bandwidth, lower and upper (1-PROBTEST) * 100% confidence limit factors and critical value for the squared coherency (e.g. arguments EDOF, BANDWIDTH, CONLWR, CONUPR and TESTCOHER) are not right near the zero and Nyquist frequencies if the PSD estimates have been smoothed by modified Daniell filters. The reason is that CROSS_SPECTRUM2 assumes that smoothing involves averaging independent frequency ordinates. This is true except near the zero and Nyquist frequencies where an average may contain contributions from negative frequencies, which are identical to and hence not independent of positive frequency spectral values. Thus, the number of degrees of freedom in PSD estimates near the 0 and Nyquist frequencies are as little as half the number of degrees of freedom of the spectral estimates away from these frequency extremes if the optional argument SMOOTH_PARAM is used.

If the optional argument SMOOTH_PARAM is used, the computed equivalent number of degrees of freedom, bandwidth, lower and upper (1-PROBTEST) * 100% confidence limit factors and critical value for the squared coherency are right for PSD estimates at frequencies

(i-1)/(L+L0) for i= (nparam+1)/2 + 1 to ((L+L0) - nparam + 1)/2

where nparam = 2 * (2+sum(SMOOTH_PARAM(:)))- 1, (e.g. for frequencies i/(L+L0) for i = (nparam+1)/2, . . . , ((L+L0)-nparam-1)/2) .

For definitions, more details and algorithm, see:

(1) **Bloomfield, P., 1976:** Fourier analysis of time series- An introduction. John Wiley and Sons, New York.

(2) **Welch, P.D., 1967:** The use of Fast Fourier Transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. IEEE trans. on audio and electroacoustics, Vol. Au-15, 2, 70-73.

(3) **Diggle, P.J., 1990:** Time series: a biostatistical introduction. Clarendon Press, Oxford.

6.28 Module_Utilities

Copyright 2024 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING GENERAL AND COMPUTING UTILITIES.

MANY OF THESE ROUTINES ARE ADAPTED AND EXTENDED FROM PUBLIC DOMAIN ROUTINES FROM Numerical Recipes.

LATEST REVISION : 11/03/2024

6.28.1 function transpose2 (mat)

Purpose

Transpose the real matrix MAT.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the real matrix MAT.

6.28.2 function transpose2 (mat)

Purpose

Transpose the complex matrix MAT.

Arguments

MAT (INPUT) complex(stnd), dimension(:,:) On entry, the complex matrix MAT.

6.28.3 function transpose2 (mat)

Purpose

Transpose the integer matrix MAT.

Arguments

MAT (INPUT) integer(i4b), dimension(:,:) On entry, the integer matrix MAT.

6.28.4 function transpose2 (mat)

Purpose

Transpose the logical matrix MAT.

Arguments

MAT (INPUT) logical(lgl), dimension(:,:) On entry, the logical matrix MAT.

6.28.5 function dot_product2 (vecx, vecy)

Purpose

Forms the dot product of two real vectors.

Arguments

VECX (INPUT) real(stnd), dimension(:) On entry, the first real vector VECX.

VECY (INPUT) real(stnd), dimension(:) On entry, the second real vector VECY.

Further Details

The dot product is computed with the first min(size(VECX) , size(VECY)) elements of the two input vectors.

6.28.6 function dot_product2 (vecx, vecy)

Purpose

Forms the dot product of two complex vectors, conjugating the first vector.

Arguments

VECX (INPUT) complex(stnd), dimension(:) On entry, the first complex vector VECX.

VECY (INPUT) complex(stnd), dimension(:) On entry, the second complex vector VECY.

Further Details

The dot product is computed with the first min(size(VECX) , size(VECY)) elements of the two input vectors.

6.28.7 function dot_product2 (vecx, vecy)

Purpose

Forms the dot product of two integer vectors.

Arguments

VECX (INPUT) integer(i4b), dimension(:) On entry, the first integer vector VECX.

VECY (INPUT) integer(i4b), dimension(:) On entry, the second integer vector VECY.

Further Details

The dot product is computed with the first min(size(VECX) , size(VECY)) elements of the two input vectors.

6.28.8 function dot_product2 (vecx, vecy)

Purpose

Forms the dot product of two logical vectors.

Arguments

VECX (INPUT) logical(lgl), dimension(:) On entry, the first logical vector VECX.

VECY (INPUT) logical(lgl), dimension(:) On entry, the second logical vector VECY.

Further Details

The dot product is computed with the first min(size(VECX) , size(VECY)) elements of the two input vectors.

6.28.9 function mmproduct (vec, mat)

Purpose

Multiplies the real vector VEC by the real matrix MAT.

Arguments

VEC (INPUT) real(stnd), dimension(:) On entry, the real vector VEC.

MAT (INPUT) real(stnd), dimension(:,:) On entry, the real matrix MAT.

Further Details

The size of VEC must be equal to the number of rows of MAT.

6.28.10 function mmproduct (mat, vec2)

Purpose

Multiplies the real matrix MAT by the real vector VEC2.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the real matrix MAT.

VEC2 (INPUT) real(stnd), dimension(:) On entry, the real vector VEC2.

Further Details

The size of VEC2 must be equal to the number of columns of MAT.

6.28.11 function mmproduct (mat1, mat2)

Purpose

Multiplies the real matrix MAT1 by the real matrix MAT2.

Arguments

MAT1 (INPUT) real(stnd), dimension(:,:) On entry, the first real matrix MAT1.

MAT2 (INPUT) real(stnd), dimension(:,:) On entry, the second real matrix MAT2.

Further Details

The number of rows of MAT2 must be equal to the number of columns of MAT1, otherwise the matrix product is computed with the first min(size(MAT1,2) , size(MAT2,1)) columns of MAT1 and rows of MAT2.

6.28.12 function mmproduct (vec, mat)

Purpose

Multiplies the complex vector VEC by the complex matrix MAT.

Arguments

VEC (INPUT) complex(stnd), dimension(:) On entry, the complex vector VEC.

MAT (INPUT) complex(stnd), dimension(:,:) On entry, the complex matrix MAT.

Further Details

The size of VEC must be equal to the number of rows of MAT.

6.28.13 function mmproduct (mat, vec2)

Purpose

Multiplies the complex matrix MAT by the complex vector VEC2.

Arguments

MAT (INPUT) complex(stnd), dimension(:,:) On entry, the complex matrix MAT.

VEC2 (INPUT) complex(stnd), dimension(:) On entry, the complex vector VEC2.

Further Details

The size of VEC2 must be equal to the number of columns of MAT.

6.28.14 function mmproduct (mat1, mat2)

Purpose

Multiplies the complex matrix MAT1 by the complex matrix MAT2

Arguments

MAT1 (INPUT) complex(stnd), dimension(:,:) On entry, the first complex matrix MAT1.

MAT2 (INPUT) complex(stnd), dimension(:,:) On entry, the second complex matrix MAT2.

Further Details

The number of rows of MAT2 must be equal to the number of columns of MAT1, otherwise the matrix product is computed with the first min(size(MAT1,2) , size(MAT2,1)) columns of MAT1 and rows of MAT2.

6.28.15 function matmul2 (vec, mat)

Purpose

Multiplies the real vector VEC by the real matrix MAT.

Arguments

VEC (INPUT) real(stnd), dimension(:) On entry, the real vector VEC.

MAT (INPUT) real(stnd), dimension(:,:) On entry, the real matrix MAT.

Further Details

The size of VEC must be equal to the number of rows of MAT.

This function will use the BLAS through the BLAS_interfaces module if the C processor macro _BLAS is activated during compilation. Alternatively, if the _OPENMP3 macro is activated during compilation, this function will be parallelized with OPENMP.

6.28.16 function matmul2 (mat, vec2)

Purpose

Multiplies the real matrix MAT by the real vector VEC2.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the real matrix MAT.

VEC2 (INPUT) real(stnd), dimension(:) On entry, the real vector VEC2.

Further Details

The size of VEC2 must be equal to the number of columns of MAT.

This function will use the BLAS through the BLAS_interfaces module if the C processor macro _BLAS is activated during compilation. Alternatively, if the _OPENMP3 macro is activated during compilation, this function will be parallelized with OPENMP.

6.28.17 `function matmul2 ( mat1, mat2 )`

Purpose

Multiplies the real matrix MAT1 by the real matrix MAT2.

Arguments

MAT1 (INPUT) real(stnd), dimension(:,:) On entry, the first real matrix MAT1.

MAT2 (INPUT) real(stnd), dimension(:,:) On entry, the second real matrix MAT2.

Further Details

The number of rows of MAT2 must be equal to the number of columns of MAT1, otherwise the matrix product is computed with the first min(size(MAT1,2) , size(MAT2,1)) columns of MAT1 and rows of MAT2.

This function will use the BLAS through the BLAS_interfaces module if the C processor macro _BLAS is activated during compilation. On the other hand, if the _BLAS macro is not activated and the _OPENMP3 macro is activated during compilation, this function will be parallelized with OPENMP if the matrices are big enough.

6.28.18 `function matmul2 ( vec, mat )`

Purpose

Multiplies the complex vector VEC by the complex matrix MAT.

Arguments

VEC (INPUT) complex(stnd), dimension(:) On entry, the complex vector VEC.

MAT (INPUT) complex(stnd), dimension(:,:) On entry, the complex matrix MAT.

Further Details

The size of VEC must be equal to the number of rows of MAT.

This function will use the BLAS through the BLAS_interfaces module if the C processor macro _BLAS is activated during compilation. Alternatively, if the _OPENMP3 macro is activated during compilation, this function will be parallelized with OPENMP.

6.28.19 `function matmul2 ( mat, vec2 )`

Purpose

Multiplies the complex matrix MAT by the complex vector VEC2.

Arguments

MAT (INPUT) complex(stnd), dimension(:,:) On entry, the complex matrix MAT.

VEC2 (INPUT) complex(stnd), dimension(:) On entry, the complex vector VEC2.

Further Details

The size of VEC2 must be equal to the number of columns of MAT.

This function will use the BLAS through the BLAS_interfaces module if the C processor macro _BLAS is activated during compilation. Alternatively, if the _OPENMP3 macro is activated during compilation, this function will be parallelized with OPENMP.

6.28.20 `function matmul2 ( mat1, mat2 )`

Purpose

Multiplies the complex matrix MAT1 by the complex matrix MAT2.

Arguments

MAT1 (INPUT) complex(stnd), dimension(:,:) On entry, the first complex matrix MAT1.

MAT2 (INPUT) complex(stnd), dimension(:,:) On entry, the second complex matrix MAT2.

Further Details

The number of rows of MAT2 must be equal to the number of columns of MAT1, otherwise the matrix product is computed with the first min(size(MAT1,2) , size(MAT2,1)) columns of MAT1 and rows of MAT2.

This function will use the BLAS through the BLAS_interfaces module if the C processor macro _BLAS is activated during compilation. On the other hand, if the _BLAS macro is not activated and the _OPENMP3 macro is activated during compilation, this function will be parallelized with OPENMP if the matrices are big enough.

6.28.21 function matmul2 (vec, mat)

Purpose

Multiplies the integer vector VEC by the integer matrix MAT.

Arguments

VEC (INPUT) integer(i4b), dimension(:) On entry, the integer vector VEC.

MAT (INPUT) integer(i4b), dimension(:,:) On entry, the integer matrix MAT.

Further Details

The size of VEC must be equal to the number of rows of MAT.

If the _OPENMP3 macro is activated during compilation, this function will be parallelized with OPENMP.

6.28.22 function matmul2 (mat, vec2)

Purpose

Multiplies the integer matrix MAT by the integer vector VEC2.

Arguments

MAT (INPUT) integer(i4b), dimension(:,:) On entry, the integer matrix MAT.

VEC2 (INPUT) integer(i4b), dimension(:) On entry, the integer vector VEC2.

Further Details

The size of VEC2 must be equal to the number of columns of MAT.

If the _OPENMP3 macro is activated during compilation, this function will be parallelized with OPENMP.

6.28.23 function matmul2 (mat1, mat2)

Purpose

Multiplies the integer matrix MAT1 by the integer matrix MAT2.

Arguments

MAT1 (INPUT) integer(i4b), dimension(:,:) On entry, the first integer matrix MAT1.

MAT2 (INPUT) integer(i4b), dimension(:,:) On entry, the second integer matrix MAT2.

Further Details

The number of rows of MAT2 must be equal to the number of columns of MAT1, otherwise the matrix product is computed with the first min(size(MAT1,2) , size(MAT2,1)) columns of MAT1 and rows of MAT2.

If the _OPENMP3 macro is activated during compilation, this function will be parallelized with OPENMP if the matrices are big enough.

6.28.24 function matmul2 (vec, mat)

Purpose

Multiplies the logical vector VEC by the logical matrix MAT.

Arguments

VEC (INPUT) logical(lgl), dimension(:) On entry, the logical vector VEC.

MAT (INPUT) logical(lgl), dimension(:,:) On entry, the logical matrix MAT.

Further Details

The size of VEC must be equal to the number of rows of MAT.

If the _OPENMP3 macro is activated during compilation, this function will be parallelized with OPENMP.

6.28.25 function matmul2 (mat, vec2)

Purpose

Multiplies the logical matrix MAT by the logical vector VEC2.

Arguments

MAT (INPUT) logical(lgl), dimension(:,:) On entry, the logical matrix MAT.

VEC2 (INPUT) logical(lgl), dimension(:) On entry, the logical vector VEC2.

Further Details

The size of VEC2 must be equal to the number of columns of MAT.

If the _OPENMP3 macro is activated during compilation, this function will be parallelized with OPENMP.

6.28.26 function matmul2 (mat1, mat2)

Purpose

Multiplies the logical matrix MAT1 by the logical matrix MAT2.

Arguments

MAT1 (INPUT) logical(lgl), dimension(:,:) On entry, the first logical matrix MAT1.

MAT2 (INPUT) logical(lgl), dimension(:,:) On entry, the second logical matrix MAT2.

Further Details

The number of rows of MAT2 must be equal to the number of columns of MAT1, otherwise the matrix product is computed with the first min(size(MAT1,2) , size(MAT2,1)) columns of MAT1 and rows of MAT2.

If the _OPENMP3 macro is activated during compilation, this function will be parallelized with OPENMP if the matrices are big enough.

6.28.27 subroutine array_copy (src, dest, n_copied, n_not_copied)

Purpose

Copies to a destination integer array DEST the one-dimensional integer array SRC, or as much of SRC as will fit in DEST.

Returns the number of components copied as N_COPIED, and the number of components not copied as N_NOT_COPIED.

Arguments

SRC (INPUT) integer(i4b), dimension(:) On entry, the integer vector SRC.

DEST (OUTPUT) integer(i4b), dimension(:) On output, the integer vector DEST.

N_COPIED (OUTPUT) integer(i4b) On output, the integer N_COPIED.

N_NOT_COPIED (OUTPUT) integer(i4b) On output, the integer N_NOT_COPIED.

6.28.28 subroutine array_copy (src, dest, n_copied, n_not_copied)

Purpose

Copies to a destination real array DEST the one-dimensional real array SRC, or as much of SRC as will fit in DEST.

Returns the number of components copied as N_COPIED, and the number of components not copied as N_NOT_COPIED.

Arguments

SRC (INPUT) real(stnd), dimension(:) On entry, the real vector SRC.

DEST (OUTPUT) real(stnd), dimension(:) On output, the real vector DEST.

N_COPIED (OUTPUT) real(stnd) On output, the real N_COPIED.

N_NOT_COPIED (OUTPUT) real(stnd) On output, the real N_NOT_COPIED.

6.28.29 subroutine array_copy (src, dest, n_copied, n_not_copied)

Purpose

Copies to a destination complex array DEST the one-dimensional complex array SRC, or as much of SRC as will fit in DEST.

Returns the number of components copied as N_COPIED, and the number of components not copied as N_NOT_COPIED.

Arguments

SRC (INPUT) complex(stnd), dimension(:) On entry, the complex vector SRC.

DEST (OUTPUT) complex(stnd), dimension(:) On output, the complex vector DEST.

N_COPIED (OUTPUT) complex(stnd) On output, the complex N_COPIED.

N_NOT_COPIED (OUTPUT) complex(stnd) On output, the complex N_NOT_COPIED.

6.28.30 subroutine swap (a, b)

Purpose

Swap the integers A and B.

Arguments

A (INPUT/OUTPUT) integer(i4b) On entry, the integer A.

B (INPUT/OUTPUT) integer(i4b) On entry, the integer B.

6.28.31 subroutine swap (a, b)

Purpose

Swap the the real numbers A and B.

Arguments

A (INPUT/OUTPUT) real(stnd) On entry, the real A.

B (INPUT/OUTPUT) real(stnd) On entry, the real B.

6.28.32 subroutine swap (a, b)

Purpose

Swap the complex numbers A and B.

Arguments

A (INPUT/OUTPUT) complex(stnd) On entry, the complex A.

B (INPUT/OUTPUT) complex (stnd) On entry, the complex B.

6.28.33 subroutine swap (a, b)

Purpose

Swap the corresponding elements of the one-dimensional integer arrays A and B.

Arguments

A (INPUT/OUTPUT) integer(i4b), dimension(:) On entry, the integer vector A.

B (INPUT/OUTPUT) integer(i4b), dimension(:) On entry, the integer vector B.

Further Details

The sizes of vectors A and B must match.

6.28.34 subroutine swap (a, b)

Purpose

Swap the corresponding elements of the one-dimensional real arrays A and B.

Arguments

A (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the real vector A.

B (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the real vector B.

Further Details

The sizes of vectors A and B must match.

6.28.35 subroutine swap (a, b)

Purpose

Swap the corresponding elements of the one-dimensional complex arrays A and B.

Arguments

A (INPUT/OUTPUT) complex(stnd), dimension(:) On entry, the complex vector A.

B (INPUT/OUTPUT) complex(stnd), dimension(:) On entry, the complex vector B.

Further Details

The sizes of vectors A and B must match.

6.28.36 subroutine swap (a, b)

Purpose

Swap the corresponding elements of the two-dimensional integer arrays A and B.

Arguments

A (INPUT/OUTPUT) integer(i4b), dimension(:,:) On entry, the integer matrix A.

B (INPUT/OUTPUT) integer(i4b), dimension(:,:) On entry, the integer matrix B.

Further Details

The shapes of matrices A and B must match.

If the _OPENMP3 macro is activated during compilation, this function will be parallelized with OPENMP if the matrices are big enough.

6.28.37 subroutine swap (a, b)

Purpose

Swap the corresponding elements of the two-dimensional real arrays A and B.

Arguments

A (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real matrix A.

B (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real matrix B.

Further Details

The shapes of matrices A and B must match.

If the _OPENMP3 macro is activated during compilation, this function will be parallelized with OPENMP if the matrices are big enough.

6.28.38 subroutine swap (a, b)

Purpose

Swap the corresponding elements of the two-dimensional complex arrays A and B.

Arguments

A (INPUT/OUTPUT) complex(stnd), dimension(:,:) On entry, the complex matrix A.

B (INPUT/OUTPUT) complex(stnd), dimension(:,:) On entry, the complex matrix B.

Further Details

The shapes of matrices A and B must match.

If the _OPENMP3 macro is activated during compilation, this function will be parallelized with OPENMP if the matrices are big enough.

6.28.39 subroutine swap (a, b, mask)

Purpose

Swap the integers A and B if MASK=true.

Arguments

A (INPUT/OUTPUT) integer(i4b) On entry, the integer A.

B (INPUT/OUTPUT) integer(i4b) On entry, the integer B.

MASK (INPUT) logical(lgl) On entry, the logical mask value.

6.28.40 subroutine swap (a, b, mask)

Purpose

Swap the reals A and B if MASK=true.

Arguments

A (INPUT/OUTPUT) real(stnd) On entry, the real A.

B (INPUT/OUTPUT) real(stnd) On entry, the real B.

MASK (INPUT) logical(lgl) On entry, the logical mask value.

6.28.41 subroutine swap (a, b, mask)

Purpose

Swap the complex A and B if MASK=true.

Arguments

A (INPUT/OUTPUT) complex(stnd) On entry, the complex A.

B (INPUT/OUTPUT) real(stnd) On entry, the complex B.

MASK (INPUT) logical(lgl) On entry, the logical mask value.

6.28.42 subroutine swap (a, b, mask)

Purpose

Swap the corresponding elements of the one-dimensional integer arrays A and B, if the corresponding element of the one-dimensional logical array MASK is true.

Arguments

A (INPUT/OUTPUT) integer(i4b), dimension(:) On entry, the integer vector A.

B (INPUT/OUTPUT) integer(i4b), dimension(:) On entry, the integer vector B.

MASK (INPUT) logical(lgl), dimension(:) On entry, the logical mask vector.

Further Details

The sizes of vectors A, B and MASK must match.

6.28.43 subroutine swap (a, b, mask)

Purpose

Swap the corresponding elements of the one-dimensional real arrays A and B, if the corresponding element of the one-dimensional logical array MASK is true.

Arguments

A (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the real vector A.

B (INPUT/OUTPUT) real(stnd), dimension(:) On entry, the real vector B.

MASK (INPUT) logical(lgl), dimension(:) On entry, the logical mask vector.

Further Details

The sizes of vectors A, B and MASK must match.

6.28.44 subroutine swap (a, b, mask)

Purpose

Swap the corresponding elements of the one-dimensional complex arrays A and B, if the corresponding element of the one-dimensional logical array MASK is true.

Arguments

A (INPUT/OUTPUT) complex(stnd), dimension(:) On entry, the complex vector A.

B (INPUT/OUTPUT) complex(stnd), dimension(:) On entry, the complex vector B.

MASK (INPUT) logical(lgl), dimension(:) On entry, the logical mask vector.

Further Details

The sizes of vectors A, B and MASK must match.

6.28.45 subroutine swap (a, b, mask)

Purpose

Swap the corresponding elements of the two-dimensional integer arrays A and B, if the corresponding element of the two-dimensional logical array MASK is true.

Arguments

A (INPUT/OUTPUT) integer(i4b), dimension(:,:) On entry, the integer matrix A.

B (INPUT/OUTPUT) integer(i4b), dimension(:,:) On entry, the integer matrix B.

MASK (INPUT) logical(lgl), dimension(:,:) On entry, the logical mask matrix.

Further Details

The shapes of matrices A, B and MASK must match.

If the _OPENMP3 macro is activated during compilation, this function will be parallelized with OPENMP if the matrices are big enough.

6.28.46 subroutine swap (a, b, mask)

Purpose

Swap the corresponding elements of the two-dimensional real arrays A and B, if the corresponding element of the two-dimensional logical array MASK is true.

Arguments

A (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real matrix A.

B (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real matrix B.

MASK (INPUT) logical(lgl), dimension(:,:) On entry, the logical mask matrix.

Further Details

The shapes of matrices A, B and MASK must match.

If the _OPENMP3 macro is activated during compilation, this function will be parallelized with OPENMP if the matrices are big enough.

6.28.47 `subroutine swap ( a, b, mask )`

Purpose

Swap the corresponding elements of the two-dimensional complex arrays A and B, if the corresponding element of the two-dimensional logical array MASK is true.

Arguments

A (INPUT/OUTPUT) complex(stnd), dimension(:,:) On entry, the complex matrix A.

B (INPUT/OUTPUT) complex(stnd), dimension(:,:) On entry, the complex matrix B.

MASK (INPUT) logical(lgl), dimension(:,:) On entry, the logical mask matrix.

Further Details

The shapes of matrices A, B and MASK must match.

If the _OPENMP3 macro is activated during compilation, this function will be parallelized with OPENMP if the matrices are big enough.

6.28.48 `subroutine mvalloc ( p, n, ialloc )`

Purpose

Reallocates an allocatable array P to an integer one-dimensional array with a new size N, while preserving its contents.

6.28.49 `subroutine mvalloc ( p, n, ialloc )`

Purpose

Reallocates an allocatable array P to a real one-dimensional array with a new size N, while preserving its contents.

6.28.50 subroutine mvalloc (p, n, ialloc)

Purpose

Reallocates an allocatable array P to a complex one-dimensional array with a new size N, while preserving its contents.

6.28.51 subroutine mvalloc (p, n, ialloc)

Purpose

Reallocates an allocatable array P to a character one-dimensional array with a new size N, while preserving its contents.

6.28.52 subroutine mvalloc (p, n, m, ialloc)

Purpose

Reallocates an allocatable array P to an integer two dimensional array with a new shape (N,M) while preserving its contents.

6.28.53 subroutine mvalloc (p, n, m, ialloc)

Purpose

Reallocates an allocatable array P to a real two dimensional array with a new shape (N,M) while preserving its contents.

6.28.54 subroutine mvalloc (p, n, m, ialloc)

Purpose

Reallocates an allocatable array P to a complex two dimensional array with a new shape (N,M) while preserving its contents.

6.28.55 function ifirstloc (mask)

Purpose

Returns the index of the first location, in a one-dimensional logical MASK, that has the value true, or returns size(MASK)+1 if all components of MASK are false .

6.28.56 function imaxloc (arr)

Purpose

Returns location of the one-dimensional integer array ARR maximum as an integer.

6.28.57 function imaxloc (arr, mask)

Purpose

Returns location as an integer of the maximum in the elements of the one-dimensional integer array ARR under the control of the one-dimensional logical array MASK. Returns size(MASK)+1 if all components of MASK are false .

6.28.58 function imaxloc (arr)

Purpose

Returns location of the one-dimensional real array ARR maximum as an integer.

6.28.59 function imaxloc (arr, mask)

Purpose

Returns location as an integer of the maximum in the elements of the one-dimensional real array ARR under the control of the one-dimensional logical array MASK. Returns size(MASK)+1 if all components of MASK are false .

6.28.60 function iminloc (arr)

Purpose

Returns location of the one-dimensional integer array ARR minimum as an integer.

6.28.61 function iminloc (arr, mask)

Purpose

Returns location as an integer of the minimum in the elements of the one-dimensional integer array ARR under the control of the one-dimensional logical array MASK. Returns size(MASK)+1 if all components of MASK are false .

6.28.62 function iminloc (arr)

Purpose

Returns location of the one-dimensional real array ARR minimum as an integer.

6.28.63 function iminloc (arr, mask)

Purpose

Returns location as an integer of the minimum in the elements of the one-dimensional real array ARR under the control of the one-dimensional logical array MASK. Returns size(MASK)+1 if all components of MASK are false .

6.28.64 subroutine assert (n1, string)

Purpose

Exit with error message STRING, if logical argument n1 is false .

6.28.65 subroutine assert (n1, n2, string)

Purpose

Exit with error message STRING, if any of the logical arguments n1, n2 are false .

6.28.66 subroutine assert (n1, n2, n3, string)

Purpose

Exit with error message STRING, if any of the logical arguments n1, n2, n3 are false .

6.28.67 subroutine assert (n1, n2, n3, n4, string)

Purpose

Exit with error message STRING, if any of the logical arguments n1, n2, n3, n4 are false .

6.28.68 subroutine assert (n, string)

Purpose

Exit with error message STRING, if any of the elements of the one-dimensional logical array N are false .

6.28.69 function assert_eq (n1, n2, string)

Purpose

Exit with error message STRING, if the integer arguments n1, n2 are not equal.

6.28.70 function assert_eq (n1, n2, n3, string)

Purpose

Exit with error message STRING, if the integer arguments n1, n2, n3 are not all equal.

6.28.71 function assert_eq (n1, n2, n3, n4, string)

Purpose

Exit with error message STRING, if the integer arguments n1, n2, n3, n4 are not all equal.

6.28.72 function assert_eq (nn, string)

Purpose

Exit with error message STRING, if the elements of the one-dimensional integer array NN are not all equal.

6.28.73 subroutine merror (string, ierror)

Purpose

Report error message STRING and optional error number IERROR and stop.

6.28.74 function arth (first, increment, n)

Purpose

Returns an one-dimensional integer array of length N containing an arithmetic progression whose first value is FIRST and whose increment is INCREMENT.

6.28.75 function arth (first, increment, n)

Purpose

Returns an one-dimensional real array of length N containing an arithmetic progression whose first value is FIRST and whose increment is INCREMENT.

6.28.76 function arth (first, increment, n)

Purpose

Returns an one-dimensional complex array of length N containing an arithmetic progression whose first value is FIRST and whose increment is INCREMENT.

6.28.77 function arth (first, increment, n)

Purpose

Returns a two-dimensional integer array containing size(FIRST) = size(INCREMENT) arithmetic progressions of length N whose first values are FIRST(:) and whose increments are INCREMENT(:).

It is assumed that the vector arguments FIRST and INCREMENT have the same length.

6.28.78 function arth (first, increment, n)

Purpose

Returns a two-dimensional real array containing size(FIRST) = size(INCREMENT) arithmetic progressions of length N whose first values are FIRST(:) and whose increments are INCREMENT(:).

It is assumed that the vector arguments FIRST and INCREMENT have the same length.

6.28.79 function arth (first, increment, n)

Purpose

Returns a two-dimensional complex array containing size(FIRST) = size(INCREMENT) arithmetic progressions of length N whose first values are FIRST(:) and whose increments are INCREMENT(:).

It is assumed that the vector arguments FIRST and INCREMENT have the same length.

6.28.80 function geop (first, factor, n)

Purpose

Returns an one-dimensional integer array of length N containing a geometric progression whose first value is FIRST and whose multiplier is FACTOR.

6.28.81 function geop (first, factor, n)

Purpose

Returns an one-dimensional real array of length N containing a geometric progression whose first value is FIRST and whose multiplier is FACTOR.

6.28.82 function geop (first, factor, n)

Purpose

Returns an one-dimensional complex array of length N containing a geometric progression whose first value is FIRST and whose multiplier is FACTOR.

6.28.83 function geop (first, factor, n)

Purpose

Returns a two-dimensional integer array containing size(FIRST) = size(FACTOR) geometric progressions of length N whose first values are FIRST(:) and whose multipliers are FACTOR(:).

It is assumed that the vector arguments FIRST and FACTOR have the same length.

6.28.84 function geop (first, factor, n)

Purpose

Returns a two-dimensional real array containing size(FIRST) = size(FACTOR) geometric progressions of length N whose first values are FIRST(:) and whose multipliers are FACTOR(:).

It is assumed that the vector arguments FIRST and FACTOR have the same length.

6.28.85 function geop (first, factor, n)

Purpose

Returns a two-dimensional complex array containing size(FIRST) = size(FACTOR) geometric progressions of length N whose first values are FIRST(:) and whose multipliers are FACTOR(:).

It is assumed that the vector arguments FIRST and FACTOR have the same length.

6.28.86 function cumsum (arr, seed)

Purpose

Returns a rank one integer array containing the cumulative sum of the rank one integer array ARR. If the optional argument SEED is present, it is added to all components of the result.

6.28.87 function cumsum (arr, seed)

Purpose

Returns a rank one real array containing the cumulative sum of the rank one real array ARR. If the optional argument SEED is present, it is added to all components of the result.

6.28.88 function cumsum (arr, seed)

Purpose

Returns a rank one complex array containing the cumulative sum of the rank one complex array ARR. If the optional argument SEED is present, it is added to all components of the result.

6.28.89 function cumprod (arr, seed)

Purpose

Returns a rank one integer array containing the cumulative product of the rank one integer array ARR. If the optional argument SEED is present, it is multiplied into all components of the result.

6.28.90 function cumprod (arr, seed)

Purpose

Returns a rank one real array containing the cumulative product of the rank one real array ARR. If the optional argument SEED is present, it is multiplied into all components of the result.

6.28.91 function cumprod (arr, seed)

Purpose

Returns a rank one complex array containing the cumulative product of the rank one complex array ARR. If the optional argument SEED is present, it is multiplied into all components of the result.

6.28.92 function poly (x, coeffs)

Purpose

Returns a real scalar containing the result of evaluating the polynomial P(X) for X real with one-dimensional real coefficient vector COEFFS

P(X) = COEFFS(1) + COEFFS(2) * X + COEFFS(3) * X**(2) + . . .

6.28.93 function poly (x, coeffs)

Purpose

Returns a complex scalar containing the result of evaluating the polynomial P(X) for X complex with one-dimensional real coefficient vector COEFFS

P(X) = COEFFS(1) + COEFFS(2) * X + COEFFS(3) * X**(2) + . . .

6.28.94 function poly (x, coeffs)

Purpose

Returns a complex scalar containing the result of evaluating the polynomial P(X) for X complex with one-dimensional complex coefficient vector COEFFS

P(X) = COEFFS(1) + COEFFS(2) * X + COEFFS(3) * X**(2) + . . .

6.28.95 function poly (x, coeffs)

Purpose

Returns a real vector containing the results of evaluating the polynomials P(X(:)) for X(:) real with one-dimensional real coefficient vector COEFFS

P(X(:)) = COEFFS(1) + COEFFS(2) * X(:) + COEFFS(3) * X(:)**(2) + ...

6.28.96 function poly (x, coeffs, mask)

Purpose

Returns a real vector containing the results of evaluating the polynomials P(X(:)) for X(:) real with one-dimensional real coefficient vector COEFFS

P(X(:)) = COEFFS(1) + COEFFS(2) * X(:) + COEFFS(3) * X(:)**(2) + ...

under the control of the logical argument MASK. If MASK(i) = false, the polynomial is not evaluated at X(i).

6.28.97 function poly_term (coeffs, x)

Purpose

Returns a real array of size(COEFFS) containing the partial cumulants of the polynomial with real coefficients COEFFS evaluated at the real scalar X. On entry, the coefficients in COEFFS are arranged from highest order to lowest-order coefficients.

6.28.98 function poly_term (coeffs, x)

Purpose

Returns a complex array of size(COEFFS) containing the partial cumulants of the polynomial with complex coefficients COEFFS evaluated at the complex scalar X. On entry, the coefficients in COEFFS are arranged from highest order to lowest-order coefficients.

6.28.99 function zroots_unity (n, nn)

Purpose

Complex function returning a complex array containing nn consecutive powers of the nth complex root of unity.

6.28.100 subroutine update_rk1 (mat, u, v)

Purpose

Updates the integer matrix MAT with the outer sum of the two integer vectors U and V :

MAT = MAT + U * V'

Arguments

MAT (INPUT/OUTPUT) integer(i4b), dimension(:,:) On entry, the integer matrix MAT.

U (INPUT) integer(i4b), dimension(:) On entry, the integer vector U.

V (INPUT) integer(i4b), dimension(:) On entry, the integer vector V.

6.28.101 subroutine update_rk1 (mat, u, v)

Purpose

Updates the real matrix MAT with the outer sum of the two reals vectors U and V :

MAT = MAT + U * V'

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real matrix MAT.

U (INPUT) real(stnd), dimension(:) On entry, the real vector U.

V (INPUT) real(stnd), dimension(:) On entry, the real vector V.

6.28.102 subroutine update_rk1 (mat, u, v)

Purpose

Updates the complex matrix MAT with the outer sum of the two complex vectors U and V :

MAT = MAT + U * V'

Arguments

MAT (INPUT/OUTPUT) complex(stnd), dimension(:,:) On entry, the complex matrix MAT.

U (INPUT) complex(stnd), dimension(:) On entry, the complex vector U.

V (INPUT) complex(stnd), dimension(:) On entry, the complex vector V.

6.28.103 subroutine update_rk2 (mat, u, v, u2, v2)

Purpose

Updates the integer matrix MAT with the outer sums of the integer vectors U, V, U2 and V2:

MAT = MAT + U * V' + U2 * V2'

Arguments

MAT (INPUT/OUTPUT) integer(i4b), dimension(:,:) On entry, the integer matrix MAT.

U (INPUT) integer(i4b), dimension(:) On entry, the integer vector U.

V (INPUT) integer(i4b), dimension(:) On entry, the integer vector V.

U2 (INPUT) integer(i4b), dimension(:) On entry, the integer vector U2.

V2 (INPUT) integer(i4b), dimension(:) On entry, the integer vector V2.

6.28.104 subroutine update_rk2 (mat, u, v, u2, v2)

Purpose

Updates the real matrix MAT with the outer sums of the real vectors U, V, U2 and V2:

MAT = MAT + U * V' + U2 * V2'

Arguments

MAT (INPUT/OUTPUT) real(stnd), dimension(:,:) On entry, the real matrix MAT.

U (INPUT) real(stnd), dimension(:) On entry, the real vector U.

V (INPUT) real(stnd), dimension(:) On entry, the real vector V.

U2 (INPUT) real(stnd), dimension(:) On entry, the real vector U2.

V2 (INPUT) real(stnd), dimension(:) On entry, the real vector V2.

6.28.105 subroutine update_rk2 (mat, u, v, u2, v2)

Purpose

Updates the complex matrix MAT with the outer sums of the complex vectors U, V, U2 and V2:

MAT = MAT + U * V' + U2 * V2'

Arguments

MAT (INPUT/OUTPUT) complex(stnd), dimension(:,:) On entry, the complex matrix MAT.

U (INPUT) complex(stnd), dimension(:) On entry, the complex vector U.

V (INPUT) complex(stnd), dimension(:) On entry, the complex vector V.

U2 (INPUT) complex(stnd), dimension(:) On entry, the complex vector U2.

V2 (INPUT) complex(stnd), dimension(:) On entry, the complex vector V2.

6.28.106 function outerprod (a, b)

Purpose

Returns a matrix that is the outer product of the two integer vectors A and B .

Arguments

A (INPUT) integer(i4b), dimension(:) On entry, the integer vector A.

B (INPUT) integer(i4b), dimension(:) On entry, the integer vector B.

6.28.107 function outerprod (a, b)

Purpose

Returns a matrix that is the outer product of the two real vectors A and B .

Arguments

A (INPUT) real(stnd), dimension(:) On entry, the real vector A.

B (INPUT) real(stnd), dimension(:) On entry, the real vector B.

6.28.108 function outerprod (a, b)

Purpose

Returns a matrix that is the outer product of the two complex vectors A and B .

Arguments

A (INPUT) complex(stnd), dimension(:) On entry, the complex vector A.

B (INPUT) complex(stnd), dimension(:) On entry, the complex vector B.

6.28.109 function outerdiv (a, b)

Purpose

Returns a matrix that is the outer quotient of the two real vectors A and B .

Arguments

A (INPUT) real(stnd), dimension(:) On entry, the real vector A.

B (INPUT) real(stnd), dimension(:) On entry, the real vector B.

Further Details

It is assumed that none of the elements of B is zero.

6.28.110 function outerdiv (a, b)

Purpose

Returns a matrix that is the outer quotient of the two complex vectors A and B .

Arguments

A (INPUT) complex(stnd), dimension(:) On entry, the complex vector A.

B (INPUT) complex(stnd), dimension(:) On entry, the complex vector B.

Further Details

It is assumed that none of the elements of B is zero.

6.28.111 function outersum (a, b)

Purpose

Returns a matrix that is the outer sum of the two integer vectors A and B .

Arguments

A (INPUT) integer(i4b), dimension(:) On entry, the integer vector A.

B (INPUT) integer(i4b), dimension(:) On entry, the integer vector B.

6.28.112 function outersum (a, b)

Purpose

Returns a matrix that is the outer sum of the two real vectors A and B .

Arguments

A (INPUT) real(stnd), dimension(:) On entry, the real vector A.

B (INPUT) real(stnd), dimension(:) On entry, the real vector B.

6.28.113 function outersum (a, b)

Purpose

Returns a matrix that is the outer sum of the two complex vectors A and B .

Arguments

A (INPUT) complex(stnd), dimension(:) On entry, the complex vector A.

B (INPUT) complex(stnd), dimension(:) On entry, the complex vector B.

6.28.114 function outerdiff (a, b)

Purpose

Returns a matrix that is the outer difference of the two integer vectors A and B .

Arguments

A (INPUT) integer(i4b), dimension(:) On entry, the integer vector A.

B (INPUT) integer(i4b), dimension(:) On entry, the integer vector B.

6.28.115 function outerdiff (a, b)

Purpose

Returns a matrix that is the outer difference of the two real vectors A and B .

Arguments

A (INPUT) real(stnd), dimension(:) On entry, the real vector A.

B (INPUT) real(stnd), dimension(:) On entry, the real vector B.

6.28.116 function outerdiff (a, b)

Purpose

Returns a matrix that is the outer difference of the two complex vectors A and B .

Arguments

A (INPUT) complex(stnd), dimension(:) On entry, the complex vector A.

B (INPUT) complex(stnd), dimension(:) On entry, the complex vector B.

6.28.117 function outerand (a, b)

Purpose

Returns a matrix that is the outer logical AND of two logical vectors A and B .

Arguments

A (INPUT) logical(lgl), dimension(:) On entry, the logical vector A.

B (INPUT) logical(lgl), dimension(:) On entry, the logical vector B.

6.28.118 function outeror (a, b)

Purpose

Returns a matrix that is the outer logical OR of two logical vectors A and B .

Arguments

A (INPUT) logical(lgl), dimension(:) On entry, the logical vector A.

B (INPUT) logical(lgl), dimension(:) On entry, the logical vector B.

6.28.119 function triangle (upper, j, k, extra)

Purpose

Return an upper (if UPPER=true) or lower (if UPPER=false) triangular logical mask.

Arguments

UPPER (INPUT) logical(lgl) On entry, the logical scalar UPPER.

J (INPUT) integer(i4b) On entry, the number of rows of the logical matrix returned by the function.

K (INPUT) integer(i4b) On entry, the number of columns of the logical matrix returned by the function.

EXTRA (INPUT) integer(i4b) On entry, an integer value to set the values of the logical matrix returned by the function. By default, EXTRA is set to zero.

6.28.120 function abse (vec)

Purpose

This function computes the 2-norm (i.e., the Euclidean norm) of the real vector VEC of length n.

Arguments

VEC (INPUT) real(stnd), dimension(:) On entry, the real vector VEC.

Further Details

The routine is based on methods from reference (1) and uses compensated summation in order to minimize rounding errors.

For more details, see:

(1) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.121 `function abse ( vec )`

Purpose

This function computes the 2-norm (i.e., the Euclidean norm) of the complex vector VEC of length n.

Arguments

VEC (INPUT) complex(stnd), dimension(:) On entry, the complex vector VEC.

Further Details

The routine is based on methods from reference (1) and uses compensated summation in order to minimize rounding errors.

For more details, see:

(1) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.122 `function abse ( mat )`

Purpose

This function computes the 2-norm (i.e., the Frobenius norm) of the real n-by-m matrix MAT.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the real matrix MAT.

Further Details

The routine is based on methods from reference (1) and uses compensated summation in order to minimize rounding errors.

For more details, see:

(1) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.123 function abse (mat)

Purpose

This function computes the 2-norm (i.e., the Frobenius norm) of the complex n-by-m matrix MAT.

Arguments

MAT (INPUT) complex(stnd), dimension(:,:) On entry, the complex matrix MAT.

Further Details

The routine is based on methods from reference (1) and uses compensated summation in order to minimize rounding errors.

For more details, see:

(1) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.124 function abse (mat, dim)

Purpose

Return the Euclidean norms of the columns (DIM=2) or the rows (DIM=1) of a real matrix MAT via the function name, so that

norm := sqrt(sum(MAT * MAT,dim=3-dim)) .

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the real matrix MAT.

DIM (INPUT) integer(i4b) On entry, if:

- DIM=1 the Euclidean norms of the rows of MAT are computed.
- DIM=2 the Euclidean norms of the columns of MAT are computed.

Further Details

The routine is based on methods from reference (1) and uses compensated summation in order to minimize rounding errors.

For more details, see:

(1) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.125 function abse (mat, dim)

Purpose

Return the Euclidean norms of the columns (DIM=2) or the rows (DIM=1) of a complex matrix MAT via the function name, so that

norm := sqrt(sum(MAT * conjg(MAT),dim=3-dim)) .

Arguments

MAT (INPUT) complex(stnd), dimension(:,:) On entry, the complex matrix MAT.

DIM (INPUT) integer(i4b) On entry, if:

- DIM=1 the Euclidean norms of the rows of MAT are computed.
- DIM=2 the Euclidean norms of the columns of MAT are computed.

Further Details

The routine is based on methods from reference (1) and uses compensated summation in order to minimize rounding errors.

For more details, see:

(1) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.126 function norm (vec)

Purpose

This function computes the 2-norm (i.e., the Euclidean norm) of the real vector VEC of length n, with due regard to avoiding overflow and underflow.

Arguments

VEC (INPUT) real(stnd), dimension(:) On entry, the real vector VEC.

Further Details

The routine is based on methods from the reference (1), but this version is written in Fortran 95/2003 and is machine independent. The algorithm is also described more comprehensively in reference (2).

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

6.28.127 function norm (vec)

Purpose

This function computes the 2-norm (i.e., the Euclidean norm) of the complex vector VEC of length n, with due regard to avoiding overflow and underflow.

Arguments

VEC (INPUT) complex(stnd), dimension(:) On entry, the complex vector VEC.

Further Details

The routine is based on methods from the reference (1), but this version is written in Fortran 95/2003 and is machine independent. The algorithm is also described more comprehensively in reference (2).

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

6.28.128 function norm (mat)

Purpose

This function computes the 2-norm (i.e., the Frobenius norm) of the real n-by-m matrix MAT, with due regard to avoiding overflow and underflow.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the real matrix MAT.

Further Details

The routine is based on methods from the reference (1), but this version is written in Fortran 95/2003 and is machine independent. The algorithm is also described more comprehensively in reference (2).

The routine is also parallelized with OpenMP if the input real matrix is sufficiently big.

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

6.28.129 `function norm ( mat )`

Purpose

This function computes the 2-norm (i.e., the Frobenius norm) of the complex n-by-m matrix MAT, with due regard to avoiding overflow and underflow.

Arguments

MAT (INPUT) complex(stnd), dimension(:,:) On entry, the complex matrix MAT.

Further Details

The routine is based on methods from the reference (1), but this version is written in Fortran 95/2003 and is machine independent. The algorithm is also described more comprehensively in reference (2).

The routine is also parallelized with OpenMP if the input complex matrix is sufficiently big.

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

6.28.130 `function norm ( mat, dim )`

Purpose

Return the Euclidean norms of the columns (DIM=2) or the rows (DIM=1) of a real matrix MAT via the function name, so that

```
norm := sqrt( sum(MAT * MAT,dim=3-dim) )
```

This is done without destructive underflow or overflow.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the real matrix MAT.

DIM (INPUT) integer(i4b) On entry, if:

- DIM=1 the Euclidean norms of the rows of MAT are computed.
- DIM=2 the Euclidean norms of the columns of MAT are computed.

Further Details

The routine is based on methods from the reference (1), but this version is written in Fortran 95/2003 and is machine independent. The algorithm is also described more comprehensively in reference (2).

The routine is also parallelized with OpenMP if the input real matrix is sufficiently big.

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

6.28.131 `function norm ( mat, dim )`

Purpose

Return the Euclidean norms of the columns (DIM=2) or the rows (DIM=1) of a complex matrix MAT via the function name, so that

norm := sqrt(sum(MAT * conjg(MAT),dim=3-dim))

This is done without destructive underflow or overflow.

Arguments

MAT (INPUT) complex(stnd), dimension(:,:) On entry, the complex matrix MAT.

DIM (INPUT) integer(i4b) On entry, if:

- DIM=1 the Euclidean norms of the rows of MAT are computed.
- DIM=2 the Euclidean norms of the columns of MAT are computed.

Further Details

The routine is based on methods from the reference (1), but this version is written in Fortran 95/2003 and is machine independent. The algorithm is also described more comprehensively in reference (2).

The routine is also parallelized with OpenMP if the input complex matrix is sufficiently big.

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

6.28.132 `subroutine lassq ( vec, scal, ssq )`

Purpose

LASSQ returns the values scl and smsq such that

(scl**(2)) * smsq = sum(VEC**(2)) + (scale**(2))*ssq

The value of ssq is assumed to be non-negative.

scale and ssq must be supplied in SCAL and SSQ and scl and smsq are overwritten on SCAL and SSQ, respectively.

Arguments

VEC (INPUT) real(stnd), dimension(:) The real vector for which a scaled sum of squares is computed.

SCAL (INPUT/OUTPUT) real(stnd) On entry, the value scale in the equation above. On exit, SCAL is overwritten with scl , the scaling factor for the sum of squares.

SSQ (INPUT/OUTPUT) real(stnd) On entry, the value ssq in the equation above. On exit, SSQ is overwritten with smsq , the basic sum of squares from which scl has been factored out.

Further Details

The routine is based on methods from the reference (1), but this version is written in Fortran 95/2003 and is machine independent. The algorithm is also described more comprehensively in reference (2).

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

6.28.133 `subroutine lassq ( vec, scal, ssq )`

Purpose

LASSQ returns the values scl and smsq such that

(scl**(2)) * smsq = dot_product(VEC, VEC) + (scale**(2)) * ssq

The value of ssq is assumed to be non-negative.

scale and ssq must be supplied in SCAL and SSQ and scl and smsq are overwritten on SCAL and SSQ, respectively.

Arguments

VEC (INPUT) complex(stnd), dimension(:) The complex vector for which a scaled sum of squares is computed.

SCAL (INPUT/OUTPUT) real(stnd) On entry, the value scale in the equation above. On exit, SCAL is overwritten with scl , the scaling factor for the sum of squares.

SSQ (INPUT/OUTPUT) real(stnd) On entry, the value ssq in the equation above. On exit, SSQ is overwritten with smsq , the basic sum of squares from which scl has been factored out.

Further Details

The routine is based on methods from the reference (1), but this version is written in Fortran 95/2003 and is machine independent. The algorithm is also described more comprehensively in reference (2).

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

6.28.134 subroutine lassq (mat, scal, ssq)

Purpose

LASSQ returns the values scl and smsq such that

(scl**(2)) * smsq = sum(MAT**(2)) + (scale**(2)) * ssq,

The value of ssq is assumed to be non-negative.

scale and ssq must be supplied in SCAL and SSQ and scl and smsq are overwritten on SCAL and SSQ respectively.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) The matrix for which a scaled sum of squares is computed.

SCAL (INPUT/OUTPUT) real(stnd) On entry, the value scale in the equation above. On exit, SCAL is overwritten with scl , the scaling factor for the sum of squares.

SSQ (INPUT/OUTPUT) real(stnd) On entry, the value ssq in the equation above. On exit, SSQ is overwritten with smsq , the basic sum of squares from which scl has been factored out.

Further Details

The routine is based on methods from the reference (1), but this version is written in Fortran 95/2003 and is machine independent. The algorithm is also described more comprehensively in reference (2).

The routine is also parallelized with OpenMP if the input real matrix is sufficiently big.

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

6.28.135 subroutine lassq (mat, scal, ssq)

Purpose

LASSQ returns the values scl and smsq such that

(scl**(2)) * smsq = sum(MAT * conjg(MAT)) + (scale**(2)) * ssq,

The value of ssq is assumed to be non-negative.

scale and ssq must be supplied in SCAL and SSQ and scl and smsq are overwritten on SCAL and SSQ respectively.

Arguments

MAT (INPUT) complex(stnd), dimension(:,:) The complex matrix for which a scaled sum of squares is computed.

SCAL (INPUT/OUTPUT) real(stnd) On entry, the value scale in the equation above. On exit, SCAL is overwritten with scl , the scaling factor for the sum of squares.

SSQ (INPUT/OUTPUT) real(stnd) On entry, the value ssq in the equation above. On exit, SSQ is overwritten with smsq , the basic sum of squares from which scl has been factored out.

Further Details

The routine is based on methods from the reference (1), but this version is written in Fortran 95/2003 and is machine independent. The algorithm is also described more comprehensively in reference (2).

The routine is also parallelized with OpenMP if the input complex matrix is sufficiently big.

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

6.28.136 `function norme ( vec )`

Purpose

This function computes the 2-norm (i.e., the Euclidean norm) of the real vector VEC of length n, with due regard to avoiding overflow and underflow.

Arguments

VEC (INPUT) real(stnd), dimension(:) On entry, the real vector VEC.

Further Details

The routine is based on methods from reference (1), but this version is written in Fortran 95/2003 and is machine/precision independent. The algorithm is also described more comprehensively in reference (2) and also uses compensated summation to improve the accuracy of the final result as suggested in reference (3).

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.137 function norme (vec)

Purpose

This function computes the 2-norm (i.e., the Euclidean norm) of the complex vector VEC of length n, with due regard to avoiding overflow and underflow.

Arguments

VEC (INPUT) complex(stnd), dimension(:) On entry, the complex vector VEC.

Further Details

The routine is based on methods from reference (1), but this version is written in Fortran 95/2003 and is machine/precision independent. The algorithm is also described more comprehensively in reference (2) and also uses compensated summation to improve the accuracy of the final result as suggested in reference (3).

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.138 function norme (mat)

Purpose

This function computes the 2-norm (i.e., the Frobenius norm) of the real n-by-m matrix MAT, with due regard to avoiding overflow and underflow.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the real matrix MAT.

Further Details

The routine is based on methods from reference (1), but this version is written in Fortran 95/2003 and is machine/precision independent. The algorithm is also described more comprehensively in reference (2) and also uses compensated summation to improve the accuracy of the final result as suggested in reference (3).

The routine is also parallelized with OpenMP if the input real matrix is sufficiently big.

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.139 `function norme ( mat )`

Purpose

This function computes the 2-norm (i.e., the Frobenius norm) of the complex n-by-m matrix MAT, with due regard to avoiding overflow and underflow.

Arguments

MAT (INPUT) complex(stnd), dimension(:,:) On entry, the complex matrix MAT.

Further Details

The routine is based on methods from reference (1), but this version is written in Fortran 95/2003 and is machine/precision independent. The algorithm is also described more comprehensively in reference (2) and also uses compensated summation to improve the accuracy of the final result as suggested in reference (3).

The routine is also parallelized with OpenMP if the input complex matrix is sufficiently big.

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.140 `function norme ( mat, dim )`

Purpose

Return the Euclidean norms of the columns (DIM=2) or the rows (DIM=1) of a real matrix MAT via the function name, so that

```
norme := sqrt( sum(MAT * MAT,dim=3-dim) )
```

This is done without destructive underflow or overflow.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the real matrix MAT.

DIM (INPUT) integer(i4b) On entry, if:

- DIM=1 the Euclidean norms of the rows of MAT are computed.
- DIM=2 the Euclidean norms of the columns of MAT are computed.

Further Details

The routine is based on methods from reference (1), but this version is written in Fortran 95/2003 and is machine/precision independent. The algorithm is also described more comprehensively in reference (2) and also uses compensated summation to improve the accuracy of the final result as suggested in reference (3).

The routine is also parallelized with OpenMP if the input real matrix is sufficiently big.

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.141 `function norme ( mat, dim )`

Purpose

Return the Euclidean norms of the columns (DIM=2) or the rows (DIM=1) of a complex matrix MAT via the function name, so that

norme := sqrt(sum(MAT * conjg(MAT),dim=3-dim))

This is done without destructive underflow or overflow.

Arguments

MAT (INPUT) complex(stnd), dimension(:,:) On entry, the complex matrix MAT.

DIM (INPUT) integer(i4b) On entry, if:

- DIM=1 the Euclidean norms of the rows of MAT are computed.
- DIM=2 the Euclidean norms of the columns of MAT are computed.

Further Details

The routine is based on methods from reference (1), but this version is written in Fortran 95/2003 and is machine/precision independent. The algorithm is also described more comprehensively in reference (2)

and also uses compensated summation to improve the accuracy of the final result as suggested in reference (3).

The routine is also parallelized with OpenMP if the input complex matrix is sufficiently big.

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.142 `subroutine lassqe ( vec, scal, ssq )`

Purpose

LASSQE returns the values scl and smsq such that

(scl**(2)) * smsq = sum(VEC**(2)) + (scale**(2))*ssq

The value of ssq is assumed to be non-negative.

scale and ssq must be supplied in SCAL and SSQ and scl and smsq are overwritten on SCAL and SSQ, respectively.

Arguments

VEC (INPUT) real(stnd), dimension(:) The real vector for which a scaled sum of squares is computed.

SCAL (INPUT/OUTPUT) real(stnd) On entry, the value scale in the equation above. On exit, SCAL is overwritten with scl , the scaling factor for the sum of squares.

SSQ (INPUT/OUTPUT) real(stnd) On entry, the value ssq in the equation above. On exit, SSQ is overwritten with smsq , the basic sum of squares from which scl has been factored out.

Further Details

The routine is based on methods from reference (1), but this version is written in Fortran 95/2003 and is machine/precision independent. The algorithm is also described more comprehensively in reference (2) and also uses compensated summation to improve the accuracy of the final result as suggested in reference (3).

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.143 subroutine lassqe (vec, scal, ssq)

Purpose

LASSQE returns the values scl and smsq such that

(scl**(2)) * smsq = dot_product(VEC, VEC) + (scale**(2)) * ssq

The value of ssq is assumed to be non-negative.

scale and ssq must be supplied in SCAL and SSQ and scl and smsq are overwritten on SCAL and SSQ, respectively.

Arguments

VEC (INPUT) complex(stnd), dimension(:) The complex vector for which a scaled sum of squares is computed.

SCAL (INPUT/OUTPUT) real(stnd) On entry, the value scale in the equation above. On exit, SCAL is overwritten with scl , the scaling factor for the sum of squares.

SSQ (INPUT/OUTPUT) real(stnd) On entry, the value ssq in the equation above. On exit, SSQ is overwritten with smsq , the basic sum of squares from which scl has been factored out.

Further Details

The routine is based on methods from reference (1), but this version is written in Fortran 95/2003 and is machine/precision independent. The algorithm is also described more comprehensively in reference (2) and also uses compensated summation to improve the accuracy of the final result as suggested in reference (3).

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.144 subroutine lassqe (mat, scal, ssq)

Purpose

LASSQE returns the values scl and smsq such that

(scl**(2)) * smsq = sum(MAT**(2)) + (scale**(2)) * ssq,

The value of ssq is assumed to be non-negative.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) The matrix for which a scaled sum of squares is computed.

SCAL (INPUT/OUTPUT) real(stnd) On entry, the value scale in the equation above. On exit, SCAL is overwritten with scl , the scaling factor for the sum of squares.

SSQ (INPUT/OUTPUT) real(stnd) On entry, the value ssq in the equation above. On exit, SSQ is overwritten with smsq , the basic sum of squares from which scl has been factored out.

Further Details

The routine is based on methods from reference (1), but this version is written in Fortran 95/2003 and is machine/precision independent. The algorithm is also described more comprehensively in reference (2) and also uses compensated summation to improve the accuracy of the final result as suggested in reference (3).

The routine is also parallelized with OpenMP if the input real matrix is sufficiently big.

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.145 `subroutine lassqe ( mat, scal, ssq )`

Purpose

LASSQE returns the values scl and smsq such that

(scl**(2)) * smsq = sum(MAT * conjg(MAT)) + (scale**(2)) * ssq,

The value of ssq is assumed to be non-negative.

scale and ssq must be supplied in SCAL and SSQ and scl and smsq are overwritten on SCAL and SSQ respectively.

Arguments

MAT (INPUT) complex(stnd), dimension(:,:) The complex matrix for which a scaled sum of squares is computed.

SCAL (INPUT/OUTPUT) real(stnd) On entry, the value scale in the equation above. On exit, SCAL is overwritten with scl , the scaling factor for the sum of squares.

SSQ (INPUT/OUTPUT) real(stnd) On entry, the value ssq in the equation above. On exit, SSQ is overwritten with smsq , the basic sum of squares from which scl has been factored out.

Further Details

The routine is based on methods from reference (1), but this version is written in Fortran 95/2003 and is machine/precision independent. The algorithm is also described more comprehensively in reference (2) and also uses compensated summation to improve the accuracy of the final result as suggested in reference (3).

The routine is also parallelized with OpenMP if the input complex matrix is sufficiently big.

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.146 `function norm2e ( vec )`

Purpose

This function computes the 2-norm (i.e., the Euclidean norm) of the real vector VEC of length n, with due regard to avoiding overflow and underflow.

Arguments

VEC (INPUT) real(stnd), dimension(:) On entry, the real vector VEC.

Further Details

The routine is based on methods from references (1) (2) and (3), but this version is written in Fortran 95/2003, is machine independent and uses compensated summation in order to minimize rounding errors.

For more details, see:

(1) **Anderson, E., 2002:** LAPACK3E - A Fortran 90-enhanced Version of LAPACK. UT-CS-02-497 (LAPACK Working Note 158). University of Tennessee, Knoxville.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS. ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.147 `function norm2e ( vec )`

Purpose

This function computes the 2-norm (i.e., the Euclidean norm) of the complex vector VEC of length n, with due regard to avoiding overflow and underflow.

Arguments

VEC (INPUT) complex(stnd), dimension(:) On entry, the complex vector VEC.

Further Details

The routine is based on methods from references (1) (2) and (3), but this version is written in Fortran 95/2003, is machine independent and uses compensated summation in order to minimize rounding errors.

For more details, see:

(1) **Anderson, E., 2002:** LAPACK3E - A Fortran 90-enhanced Version of LAPACK. UT-CS-02-497 (LAPACK Working Note 158). University of Tennessee, Knoxville.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS. ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.148 `function norm2e ( mat )`

Purpose

This function computes the 2-norm (i.e., the Frobenius norm) of the real n-by-m matrix MAT, with due regard to avoiding overflow and underflow.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the real matrix MAT.

Further Details

The routine is based on methods from references (1) (2) and (3), but this version is written in Fortran 95/2003, is machine independent and uses compensated summation in order to minimize rounding errors.

The routine is also parallelized with OpenMP if the input real matrix is sufficiently big.

For more details, see:

(1) **Anderson, E., 2002:** LAPACK3E - A Fortran 90-enhanced Version of LAPACK. UT-CS-02-497 (LAPACK Working Note 158). University of Tennessee, Knoxville.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS. ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.149 function norm2e (mat)

Purpose

This function computes the 2-norm (i.e., the Frobenius norm) of the complex n-by-m matrix MAT, with due regard to avoiding overflow and underflow.

Arguments

MAT (INPUT) complex(stnd), dimension(:,:) On entry, the complex matrix MAT.

Further Details

The routine is based on methods from references (1) (2) and (3), but this version is written in Fortran 95/2003, is machine independent and uses compensated summation in order to minimize rounding errors.

The routine is also parallelized with OpenMP if the input complex matrix is sufficiently big.

For more details, see:

(1) **Anderson, E., 2002:** LAPACK3E - A Fortran 90-enhanced Version of LAPACK. UT-CS-02-497 (LAPACK Working Note 158). University of Tennessee, Knoxville.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS. ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.150 function norm2e (mat, dim)

Purpose

Return the Euclidean norms of the columns (DIM=2) or the rows (DIM=1) of a real matrix MAT via the function name, so that

norm := sqrt(sum(MAT * MAT,dim=3-dim))

This is done without destructive underflow or overflow.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) On entry, the real matrix MAT.

DIM (INPUT) integer(i4b) On entry, if:

- DIM=1 the Euclidean norms of the rows of MAT are computed.
- DIM=2 the Euclidean norms of the columns of MAT are computed.

Further Details

The routine is based on methods from references (1) (2) and (3), but this version is written in Fortran 95/2003, is machine independent and uses compensated summation in order to minimize rounding errors.

The routine is also parallelized with OpenMP if the input real matrix is sufficiently big.

For more details, see:

(1) **Anderson, E., 2002:** LAPACK3E - A Fortran 90-enhanced Version of LAPACK. UT-CS-02-497 (LAPACK Working Note 158). University of Tennessee, Knoxville.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS. ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.151 `function norm2e ( mat, dim )`

Purpose

Return the Euclidean norms of the columns (DIM=2) or the rows (DIM=1) of a complex matrix MAT via the function name, so that

norm := sqrt(sum(MAT * conjg(MAT),dim=3-dim))

This is done without destructive underflow or overflow.

Arguments

MAT (INPUT) complex(stnd), dimension(:,:) On entry, the complex matrix MAT.

DIM (INPUT) integer(i4b) On entry, if:

- DIM=1 the Euclidean norms of the rows of MAT are computed.
- DIM=2 the Euclidean norms of the columns of MAT are computed.

Further Details

The routine is based on methods from references (1) (2) and (3), but this version is written in Fortran 95/2003, is machine independent and uses compensated summation in order to minimize rounding errors.

The routine is also parallelized with OpenMP if the input complex matrix is sufficiently big.

For more details, see:

(1) **Anderson, E., 2002:** LAPACK3E - A Fortran 90-enhanced Version of LAPACK. UT-CS-02-497 (LAPACK Working Note 158). University of Tennessee, Knoxville.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS. ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.152 subroutine lassq2e (vec, scal, ssq)

Purpose

LASSQ2E returns the values scl and smsq such that

(scl**(2)) * smsq = sum(VEC**(2)) + (scale**(2))*ssq,

The value of ssq is assumed to be non-negative.

scale and ssq must be supplied in SCAL and SSQ and scl and smsq are overwritten on SCAL and SSQ respectively.

Arguments

VEC (INPUT) real(stnd), dimension(:) The real vector for which a scaled sum of squares is computed.

SCAL (INPUT/OUTPUT) real(stnd) On entry, the value scale in the equation above. On exit, SCAL is overwritten with scl , the scaling factor for the sum of squares.

SSQ (INPUT/OUTPUT) real(stnd) On entry, the value ssq in the equation above. On exit, SSQ is overwritten with smsq , the basic sum of squares from which scl has been factored out.

Further Details

The routine is based on methods from references (1) (2) and (3), but this version is written in Fortran 95/2003, is machine independent and uses compensated summation in order to minimize rounding errors.

For more details, see:

(1) **Anderson, E., 2002:** LAPACK3E - A Fortran 90-enhanced Version of LAPACK. UT-CS-02-497 (LAPACK Working Note 158). University of Tennessee, Knoxville.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS. ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.153 subroutine lassq2e (vec, scal, ssq)

Purpose

LASSQ2E returns the values scl and smsq such that

(scl**(2)) * smsq = dot_product(VEC, VEC) + (scale**(2)) * ssq,

The value of ssq is assumed to be non-negative.

scale and ssq must be supplied in SCAL and SSQ and scl and smsq are overwritten on SCAL and SSQ respectively.

Arguments

VEC (INPUT) complex(stnd), dimension(:) The complex vector for which a scaled sum of squares is computed.

SCAL (INPUT/OUTPUT) real(stnd) On entry, the value scale in the equation above. On exit, SCAL is overwritten with scl , the scaling factor for the sum of squares.

SSQ (INPUT/OUTPUT) real(stnd) On entry, the value ssq in the equation above. On exit, SSQ is overwritten with smsq , the basic sum of squares from which scl has been factored out.

Further Details

The routine is based on methods from references (1) (2) and (3), but this version is written in Fortran 95/2003, is machine independent and uses compensated summation in order to minimize rounding errors.

For more details, see:

(1) **Anderson, E., 2002:** LAPACK3E - A Fortran 90-enhanced Version of LAPACK. UT-CS-02-497 (LAPACK Working Note 158). University of Tennessee, Knoxville.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS. ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.154 `subroutine lassq2e ( mat, scal, ssq )`

Purpose

LASSQ2E returns the values scl and smsq such that

(scl**(2)) * smsq = sum(MAT**(2)) + (scale**(2)) * ssq,

The value of ssq is assumed to be non-negative.

scale and ssq must be supplied in SCAL and SSQ and scl and smsq are overwritten on SCAL and SSQ respectively.

Arguments

MAT (INPUT) real(stnd), dimension(:,:) The matrix for which a scaled sum of squares is computed.

SCAL (INPUT/OUTPUT) real(stnd) On entry, the value scale in the equation above. On exit, SCAL is overwritten with scl , the scaling factor for the sum of squares.

SSQ (INPUT/OUTPUT) real(stnd) On entry, the value ssq in the equation above. On exit, SSQ is overwritten with smsq , the basic sum of squares from which scl has been factored out.

Further Details

The routine is based on methods from references (1) (2) and (3), but this version is written in Fortran 95/2003, is machine independent and uses compensated summation in order to minimize rounding errors.

The routine is also parallelized with OpenMP if the input real matrix is sufficiently big.

For more details, see:

(1) **Anderson, E., 2002:** LAPACK3E - A Fortran 90-enhanced Version of LAPACK. UT-CS-02-497 (LAPACK Working Note 158). University of Tennessee, Knoxville.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS. ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.155 `subroutine lassq2e ( mat, scal, ssq )`

Purpose

LASSQ2E returns the values scl and smsq such that

(scl**(2)) * smsq = sum(MAT * conjg(MAT)) + (scale**(2)) * ssq,

The value of ssq is assumed to be non-negative.

scale and ssq must be supplied in SCAL and SSQ and scl and smsq are overwritten on SCAL and SSQ respectively.

Arguments

MAT (INPUT) complex(stnd), dimension(:,:) The complex matrix for which a scaled sum of squares is computed.

SCAL (INPUT/OUTPUT) real(stnd) On entry, the value scale in the equation above. On exit, SCAL is overwritten with scl , the scaling factor for the sum of squares.

SSQ (INPUT/OUTPUT) real(stnd) On entry, the value ssq in the equation above. On exit, SSQ is overwritten with smsq , the basic sum of squares from which scl has been factored out.

Further Details

The routine is based on methods from references (1) (2) and (3), but this version is written in Fortran 95/2003, is machine independent and uses compensated summation in order to minimize rounding errors.

The routine is also parallelized with OpenMP if the input complex matrix is sufficiently big.

For more details, see:

(1) **Anderson, E., 2002:** LAPACK3E - A Fortran 90-enhanced Version of LAPACK. UT-CS-02-497 (LAPACK Working Note 158). University of Tennessee, Knoxville.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS. ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

(3) **Hanson, R.J., and Hopkins, T., 2018:** Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. ACM Trans. Math. Softw., Vol. 44, No 3, Article 24, 1-23.

6.28.156 subroutine scatter_add (dest, source, dest_index)

Purpose

Adds each component of the integer vector SOURCE into a component of the integer vector DEST specified by the index vector DEST_INDEX.

6.28.157 subroutine scatter_add (dest, source, dest_index)

Purpose

Adds each component of the real vector SOURCE into a component of the real vector DEST specified by the index vector DEST_INDEX.

6.28.158 subroutine scatter_add (dest, source, dest_index)

Purpose

Adds each component of the complex vector SOURCE into a component of the complex vector DEST specified by the index vector DEST_INDEX.

6.28.159 subroutine scatter_max (dest, source, dest_index)

Purpose

Takes the max operation between each component of the real vector SOURCE and a component of the real vector DEST specified by the index vector DEST_INDEX, replacing the component of DEST with the value obtained.

6.28.160 subroutine scatter_max (dest, source, dest_index)

Purpose

Takes the max operation between each component of the integer vector SOURCE and a component of the integer vector DEST specified by the index vector DEST_INDEX, replacing the component of DEST with the value obtained.

6.28.161 subroutine diagadd (mat, diag)

Purpose

Adds real vector DIAG to the diagonal of real matrix MAT.

6.28.162 subroutine diagadd (mat, diag)

Purpose

Adds complex vector DIAG to the diagonal of complex matrix MAT.

6.28.163 subroutine diagadd (mat, diag)

Purpose

Adds real scalar DIAG to the diagonal of real matrix MAT.

6.28.164 subroutine diagadd (mat, diag)

Purpose

Adds complex scalar DIAG to the diagonal of complex matrix MAT.

6.28.165 subroutine diagmult (mat, diag)

Purpose

Multiplies real vector DIAG into the diagonal of real matrix MAT.

6.28.166 subroutine diagmult (mat, diag)

Purpose

Multiplies complex vector DIAG into the diagonal of complex matrix MAT.

6.28.167 subroutine diagmult (mat, diag)

Purpose

Multiplies real scalar DIAG into the diagonal of real matrix MAT.

6.28.168 subroutine diagmult (mat, diag)

Purpose

Multiplies complex scalar DIAG into the diagonal of complex matrix MAT.

6.28.169 function get_diag (mat)

Purpose

Returns as a vector the diagonal of real matrix MAT.

6.28.170 function get_diag (mat)

Purpose

Returns as a vector the diagonal of complex matrix MAT.

6.28.171 subroutine put_diag (diag, mat)

Purpose

Set the diagonal of real matrix MAT to the values of the real vector DIAG.

6.28.172 subroutine put_diag (diag, mat)

Purpose

Set the diagonal of complex matrix MAT to the values of the complex vector DIAG.

6.28.173 subroutine put_diag (diag, mat)

Purpose

Set the diagonal of real matrix MAT to the value of the real scalar DIAG.

6.28.174 subroutine put_diag (diag, mat)

Purpose

Set the diagonal of complex matrix MAT to the value of the complex scalar DIAG.

6.28.175 subroutine unit_matrix (mat)

Purpose

Set the real matrix MAT to be a unit real matrix (if it is square).

6.28.176 subroutine unit_matrix (mat)

Purpose

Set the complex matrix MAT to be a unit complex matrix (if it is square).

6.28.177 subroutine lascl (x, cfrom, cto)

Purpose

LASCL multiplies the real scalar X by the real scalar CTO/CFROM . This is done without over/underflow as long as the final result CTO * X/CFROM does not over/underflow.

CFROM must be nonzero.

Arguments

X (INPUT/OUTPUT) real(stnd) The real to be multiplied by CTO/CFROM.

CFROM, CTO (INPUT) real(stnd) The real X is multiplied by CTO/CFROM.

Further Details

This subroutine is adapted from the routine DLASCL in LAPACK (version 3.10) with improvements suggested by E. Anderson. See:

(1) **Anderson, E., 2002:** LAPACK3E – A Fortran90-enhanced version of LAPACK. Lapack Working Note 158, University of Tennessee.

6.28.178 `subroutine lascl ( x, cfrom, cto )`

Purpose

LASCL multiplies the real vector X by the real scalar CTO/CFROM . This is done without over/underflow as long as the final result CTO * X(i)/CFROM does not over/underflow for i = 1 to size(X).

CFROM must be nonzero.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:) The real vector to be multiplied by CTO/CFROM.

CFROM, CTO (INPUT) real(stnd) The real vector X is multiplied by CTO/CFROM.

Further Details

This subroutine is adapted from the routine DLASCL in LAPACK (version 3.10) with improvements suggested by E. Anderson. See:

(1) **Anderson, E., 2002:** LAPACK3E – A Fortran90-enhanced version of LAPACK. Lapack Working Note 158, University of Tennessee.

6.28.179 `subroutine lascl ( x, cfrom, cto )`

Purpose

LASCL multiplies the real matrix X by the real scalar CTO/CFROM . This is done without over/underflow as long as the final result CTO * X(i,j)/CFROM does not over/underflow for i = 1 to size(X, 1) and j = 1 to size(X, 2).

CFROM must be nonzero.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:,:) The real matrix to be multiplied by CTO/CFROM.

CFROM, CTO (INPUT) real(stnd) The real matrix X is multiplied by CTO/CFROM.

Further Details

This subroutine is adapted from the routine DLASCL in LAPACK (version 3.10) with improvements suggested by E. Anderson. See:

(1) **Anderson, E., 2002:** LAPACK3E – A Fortran90-enhanced version of LAPACK. Lapack Working Note 158, University of Tennessee.

6.28.180 `subroutine lascl ( x, cfrom, cto, type )`

Purpose

LASCL multiplies the real matrix X by the real scalar CTO/CFROM . This is done without over/underflow as long as the final result CTO * X(i,j)/CFROM does not over/underflow for i = 1 to size(X, 1) and j = 1 to size(X, 2).

CFROM must be nonzero.

TYPE specifies that X may be full, upper triangular, lower triangular or upper Hessenberg.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:,:) The real matrix to be multiplied by CTO/CFROM.

CFROM, CTO (INPUT) real(stnd) The real matrix X is multiplied by CTO/CFROM.

TYPE (INPUT) character*1 TYPE indices the storage type of the input matrix. = 'L' or 'l': X is a lower triangular matrix. = 'U' or 'u': X is a upper triangular matrix. = 'H' or 'h': X is a upper Hessenberg matrix. = 'G' or 'g': X is a full matrix. = any other character: X is assumed to be a full matrix.

Further Details

This subroutine is adapted from the routine DLASCL in LAPACK (version 3.10) with improvements suggested by E. Anderson. See:

(1) **Anderson, E., 2002:** LAPACK3E – A Fortran90-enhanced version of LAPACK. Lapack Working Note 158, University of Tennessee.

6.28.181 `subroutine lascl ( x, cfrom, cto, mask )`

Purpose

LASCL multiplies the real scalar X by the real scalar CTO/CFROM under the control of the logical argument MASK . This is done without over/underflow as long as the final result CTO * X/CFROM does not over/underflow.

CFROM must be nonzero.

Arguments

X (INPUT/OUTPUT) real(stnd) The real to be multiplied by CTO/CFROM.

CFROM, CTO (INPUT) real(stnd) The real X is multiplied by CTO/CFROM if MASK=true.

MASK (INPUT) logical(lgl) The logical mask : if MASK=true the multiplication is done, otherwise X is left unchanged.

Further Details

This subroutine is adapted from the routine DLASCL in LAPACK (version 3.10) with improvements suggested by E. Anderson. See:

(1) **Anderson, E., 2002:** LAPACK3E – A Fortran90-enhanced version of LAPACK. Lapack Working Note 158, University of Tennessee.

6.28.182 `subroutine lascl ( x, cfrom, cto, mask )`

Purpose

LASCL multiplies the real vector X by the real scalar CTO/CFROM under the control of the logical argument MASK . This is done without over/underflow as long as the final result CTO * X(i)/CFROM does not over/underflow for i = 1 to size(X).

CFROM must be nonzero.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:) The real vector to be multiplied by CTO/CFROM.

CFROM, CTO (INPUT) real(stnd) The real X(i) is multiplied by CTO/CFROM if MASK(i)=true.

MASK (INPUT) logical(lgl), dimension(:) The logical mask : if MASK(i)=true the multiplication is done, otherwise X(i) is left unchanged.

Further Details

This subroutine is adapted from the routine DLASCL in LAPACK (version 3.10) with improvements suggested in reference (1).

The sizes of X and MASK must match.

For further details, see:

(1) **Anderson, E., 2002:** LAPACK3E – A Fortran90-enhanced version of LAPACK. Lapack Working Note 158, University of Tennessee.

6.28.183 `subroutine lascl ( x, cfrom, cto, mask )`

Purpose

LASCL multiplies the real matrix X by the real scalar CTO/CFROM under the control of the logical argument MASK . This is done without over/underflow as long as the final result CTO * X(i,j)/CFROM does not over/underflow for i = 1 to size(X, 1) and j = 1 to size(X, 2).

CFROM must be nonzero.

Arguments

X (INPUT/OUTPUT) real(stnd), dimension(:,:) The real matrix to be multiplied by CTO/CFROM.

CFROM, CTO (INPUT) real(stnd) The real X(i,j) is multiplied by CTO/CFROM if MASK(i,j)=true.

MASK (INPUT) logical(lgl), dimension(:,:) The logical mask : if MASK(i,j)=true the multiplication is done, otherwise X(i,j) is left unchanged.

Further Details

This subroutine is adapted from the routine DLASCL in LAPACK (version 3.10) with improvements suggested in reference (1).

The shapes of X and MASK must match.

For further details, see:

(1) **Anderson, E., 2002:** LAPACK3E – A Fortran90-enhanced version of LAPACK. Lapack Working Note 158, University of Tennessee.

6.28.184 `function pythag ( a, b )`

Purpose

Computes sqrt(a * a + b * b) without destructive underflow or overflow.

Arguments

A (INPUT) real(stnd) On entry, the real scalar a.

B (INPUT) real(stnd) On entry, the real scalar b.

6.28.185 `function pythage ( a, b )`

Purpose

Computes sqrt(a * a + b * b) without destructive underflow or overflow using the Blue's scaling method.

Arguments

A (INPUT) real(stnd) On entry, the real scalar a.

B (INPUT) real(stnd) On entry, the real scalar b.

Further Details

The routine is based on methods from the reference (1), but this version is written in Fortran 95/2003. It is machine independent. The algorithm is also described more comprehensively in reference (2).

For more details, see:

(1) **Blue, J.L., 1978:** A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Soft., Vol. 4, No 1, 15-23.

(2) **Anderson, E., 2018:** Algorithm 978: Safe scaling in the level 1 BLAS ACM Trans. Math. Soft., Vol. 44, No 1, Article 12, 1-28.

6.29 Module_Utilities_With_Pnter

Copyright 2018 IRD

This file is part of statpack.

statpack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

statpack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can find a copy of the GNU Lesser General Public License in the statpack/doc directory.

Authors: Pascal Terray (LOCEAN/IPSL, Paris, France)

MODULE EXPORTING UTILITIES TO MANIPULATE FORTRAN90 POINTERS.

THESE ROUTINES ARE ADAPTED AND EXTENDED FROM PUBLIC DOMAIN ROUTINES FROM Numerical Recipes.

LATEST REVISION : 21/03/2018

6.29.1 `function reallocate ( p, n )`

Purpose

Reallocates a pointer P to an integer one dimensional array with a new size N, while preserving its contents. The pointer P is deallocated on return.

Arguments

P integer(i4b), dimension(:), pointer On entry, an allocated pointer to an integer vector.

On exit, the pointer is deallocated.

N (INPUT) integer(i4b) The size N of the new pointer.

6.29.2 function reallocate (p, n)

Purpose

Reallocates a pointer P to a real one dimensional array with a new size N, while preserving its contents. The pointer P is deallocated on return.

Arguments

P real(stnd), dimension(:), pointer On entry, an allocated pointer to a real vector.

On exit, the pointer is deallocated.

N (INPUT) integer(i4b) The size N of the new pointer.

6.29.3 function reallocate (p, n)

Purpose

Reallocates a pointer P to a complex one dimensional array with a new size N, while preserving its contents. The pointer P is deallocated on return.

Arguments

P complex(stnd), dimension(:), pointer On entry, an allocated pointer to a complex vector.

On exit, the pointer is deallocated.

N (INPUT) integer(i4b) The size N of the new pointer.

6.29.4 function reallocate (p, n)

Purpose

Reallocates a pointer P to a character one dimensional array with a new size N, while preserving its contents. The pointer P is deallocated on return.

Arguments

P character(1), dimension(:), pointer On entry, an allocated pointer to a character vector.

On exit, the pointer is deallocated.

N (INPUT) integer(i4b) The size N of the new pointer.

6.29.5 function reallocate (p, n, m)

Purpose

Reallocates a pointer P to an integer two dimensional array with a new shape (N,M) while preserving its contents. The pointer P is deallocated on return.

Arguments

P integer(i4b), dimension(:,:), pointer On entry, an allocated pointer to an integer matrix.

On exit, the pointer is deallocated.

N, M (INPUT) integer(i4b) The shape (N,M) of the new pointer.

6.29.6 function reallocate (p, n, m)

Purpose

Reallocates a pointer P to a real two dimensional array with a new shape (N,M) while preserving its contents. The pointer P is deallocated on return.

Arguments

P real(stnd), dimension(:,:), pointer On entry, an allocated pointer to a real matrix.

On exit, the pointer is deallocated.

N, M (INPUT) integer(i4b) The shape (N,M) of the new pointer.

6.29.7 function reallocate (p, n, m)

Purpose

Reallocates a pointer P to a complex two dimensional array with a new shape (N,M) while preserving its contents. The pointer P is deallocated on return.

Arguments

P complex(stnd), dimension(:,:), pointer On entry, an allocated pointer to a complex matrix.

On exit, the pointer is deallocated.

N, M (INPUT) integer(i4b) The shape (N,M) of the new pointer.

6.29.8 subroutine realloc (p, n, ialloc)

Purpose

Reallocates a pointer P to an integer one dimensional array with a new size N, while preserving its contents.

Arguments

P integer(i4b), dimension(:), pointer On entry, an allocated pointer to an integer vector.

On exit, the allocated pointer with a new size of N.

N (INPUT) integer(i4b) The new size N of the pointer.

IALLOC (OUTPUT) integer On exit, IALLOC = 0 indicates successful exit. Any other values indicate an allocation problem.

6.29.9 subroutine realloc (p, n, ialloc)

Purpose

Reallocates a pointer P to a real one dimensional array with a new size N, while preserving its contents.

Arguments

P real(stnd), dimension(:), pointer On entry, an allocated pointer to a real vector.

On exit, the allocated pointer with a new size of N.

N (INPUT) integer(i4b) The new size N of the pointer.

IALLOC (OUTPUT) integer On exit, IALLOC = 0 indicates successful exit. Any other values indicate an allocation problem.

6.29.10 subroutine realloc (p, n, ialloc)

Purpose

Reallocates a pointer P to a complex one dimensional array with a new size N, while preserving its contents.

Arguments

P complex(stnd), dimension(:), pointer On entry, an allocated pointer to a complex vector.

On exit, the allocated pointer with a new size of N.

N (INPUT) integer(i4b) The new size N of the pointer.

IALLOC (OUTPUT) integer On exit, IALLOC = 0 indicates successful exit. Any other values indicate an allocation problem.

6.29.11 subroutine realloc (p, n, ialloc)

Purpose

Reallocates a pointer P to a character one dimensional array with a new size N, while preserving its contents.

Arguments

P character(1), dimension(:), pointer On entry, an allocated pointer to a character vector.

On exit, the allocated pointer with a new size of N.

N (INPUT) integer(i4b) The new size N of the pointer.

IALLOC (OUTPUT) integer On exit, IALLOC = 0 indicates successful exit. Any other values indicate an allocation problem.

6.29.12 subroutine realloc (p, n, m, ialloc)

Purpose

Reallocates a pointer P to an integer two dimensional array with a new shape (N,M) while preserving its contents.

Arguments

P integer(i4b), dimension(:,:), pointer On entry, an allocated pointer to an integer matrix.

On exit, the allocated pointer with a new shape (N,M).

N, M (INPUT) integer(i4b) The new shape (N,M) of the pointer.

IALLOC (OUTPUT) integer On exit, IALLOC = 0 indicates successful exit. Any other values indicate an allocation problem.

6.29.13 subroutine realloc (p, n, m, ialloc)

Purpose

Reallocates a pointer P to a real two dimensional array with a new shape (N,M) while preserving its contents.

Arguments

P real(stnd), dimension(:,:), pointer On entry, an allocated pointer to a real matrix.

On exit, the allocated pointer with a new shape (N,M).

N, M (INPUT) integer(i4b) The new shape (N,M) of the pointer.

IALLOC (OUTPUT) integer On exit, IALLOC = 0 indicates successful exit. Any other values indicate an allocation problem.

6.29.14 subroutine realloc (p, n, m, ialloc)

Purpose

Reallocates a pointer P to a complex two dimensional array with a new shape (N,M) while preserving its contents.

Arguments

P complex(stnd), dimension(:,:), pointer On entry, an allocated pointer to a complex matrix.

On exit, the allocated pointer with a new shape (N,M).

N, M (INPUT) integer(i4b) The new shape (N,M) of the pointer.

IALLOC (OUTPUT) integer On exit, IALLOC = 0 indicates successful exit. Any other values indicate an allocation problem.

BIBLIOGRAPHY

[accelerate] *Accelerate – Apple Accelerate framework* https://developer.apple.com/documentation/accelerate/

[aocl] *AOCL – AMD Optimizing CPU Libraries* https://www.amd.com/en/developer/aocl.html

[arm] *ARM – Arm Performance Libraries* https://developer.arm.com/downloads/-/arm-performance-libraries

[atlas] *ATLAS – Automatically Tuned Linear Algebra Software* http://math-atlas.sourceforge.net/

[blas] *BLAS – Basic Linear Algebra Subprograms* http://www.netlib.org/blas/#_presentation

[blis] *BLIS – BLAS-like Library Instantiation Software Framework* https://github.com/flame/blis

[gotoblas] *GotoBLAS* https://github.com/ianmartin/GotoBLAS-2/blob/master/01Readme.txt

[gsl] *GSL – GNU Scientific Library* https://www.gnu.org/software/gsl/

[lapack] *LAPACK – Linear Algebra PACKage* http://www.netlib.org/lapack/

[libflame] *LIBFLAME – High-performance object-based library for DLA computations* https://github.com/flame/libflame

[mkl] *MKL – Intel Math Kernel Library (Intel MKL)* http://software.intel.com/en-us/intel-mkl/

[mpi] *MPI – Message Passing Interface* http://www.mcs.anl.gov/research/projects/mpi/

[nag] *Nag – Numerical Algorithm Group Library (nAG)* https://support.nag.com/numeric/nl/nagdoc_latest/

[ncstat] *NCSTAT – NetCDF Statistical operators* https://pagesperso.locean-ipsl.upmc.fr/terray/ncstat2.2/index.html

[netcdf] *NetCDF – network Common Data Form* http://www.unidata.ucar.edu/software/netcdf/

[netcdf-f90] *NetCDF Fortran 90 Interface* https://docs.unidata.ucar.edu/netcdf-fortran/current/f90_The-NetCDF-Fortran-90-Interface-Guide.html

[nvblas] *NVBLAS - drop-in BLAS replacement, multi-GPUs accelerated subroutines* https://docs.nvidia.com/cuda/nvblas/

[openblas] *OpenBlas – An optimized BLAS library* http://www.openblas.net/

[openmp] *OpenMP – OpenMP Application Program Interface* http://www.openmp.org

[pblas] *PBLAS – Parallel Basic Linear Algebra Subprograms* http://www.netlib.org/scalapack/pblas_qref.html

[plasma] *PLASMA – Parallel Linear Algebra Software for Multicore Architectures* https://github.com/icl-utk-edu/plasma/

[qpblas] *QPBLAS - Quadruple Precision BLAS Routines* https://ccse.jaea.go.jp/software/QPBLAS/index_eng.html

[scalapack] *ScaLAPACK – Scalable LAPACK* http://www.netlib.org/scalapack/

[xblas] *XBLAS - Extra Precise Basic Linear Algebra Subroutines* https://www.netlib.org/xblas

[Abalenkovs_etal:2017] Abalenkovs, M., Bagherpour, N., Dongarra, J., Gates, .M., Haidar, A., Kurzak, J., Luszczek, P., Relton, S., Sistek, J., Stevens, D., Wu, P., Yamazaki, I., Asim YarKhan, A., and Zounon, M., (2017a) *PLASMA 17.1 Functionality Report* LAPACK Working Notes 293 (lawn293 and UT-EECS-17-751). Available at: http://www.netlib.org/lapack/lawnspdf/lawn293.pdf

[Abramowitz_Stegun:1970] Abramowitz, M., and Stegun, I.A., (1970) *Handbook of Mathematical Functions.* New York, Dover Publications.

[Anda_Park:1994] Anda, A.A. and Park, H., (1994) *Fast plane rotations with dynamic scaling.* Siam J. Matrix Anal. Appl., 15, 162-174. DOI: https://doi.org/10.1137/S0895479890193017

[Anderson_Fahey:1997] Anderson, E., and Fahey, M., (1997) *Performance improvements to LAPACK for the Cray Scientific Library.* LAPACK Working Note No 126. Available at: http://www.netlib.org/lapack/lawnspdf/lawn126.pdf

[Anderson_etal:1999] Anderson, E., Bai, Z., Bischof, C., Blacford, S., Demmel, J., Dongarra, J., Croz, J.D., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D., (1999) *LAPACK User's Guide.* 3rd Ed. SIAM, Philadelphia, PA. DOI: https://doi.org/10.1137/1.9780898719604

[Anderson:2002] Anderson, E., (2002) *LAPACK3E - A Fortran 90-enhanced Version of LAPACK.* LAPACK Working Note No 158. Available at: http://www.netlib.org/lapack/lawnspdf/lawn158.pdf

[Anderson:2018] Anderson, E., (2018) *Algorithm 978: Safe scaling in the level 1 BLAS.* ACM Trans. Math. Soft., 44:1, Article 12, 1-28. DOI: https://doi.org/10.1145/3061665

[Arbuckle_Friendly:1977] Arbuckle, J., and Friendly, M.L., (1977) *On rotating to smooth functions.* Psychometrika, 42, 127-140. DOI: https://doi.org/10.1007/BF02293749

[Bailey:1990] Bailey, D., (1990) *FFTs in External or Hierarchical Memory.* The Journal of Supercomputing, 4:1, 23-35. DOI: https://doi.org/10.1007/BF00162341

[Barlow_etal:2005] Barlow, J.L., Bosner, N., and Drmac, Z., (2005) *A new stable bidiagonal reduction algorithm.* Linear Algebra Appl., 397:1, 35-84. DOI: https://doi.org/10.1016/j.laa.2004.09.019

[Barnard:1978] Barnard, J., (1978) *Algorithm AS126: Probability Integral of the normal range.* Appl. Statist., 27:2, 197-198. DOI: https://doi.org/10.2307/2346956

[Berry_etal:1990] Berry, K.J., Mielke, P.W., and Cran, G.W., (1990) *Algorithm AS R83: A remark on Algorithm AS 109: Inverse of the Incomplete Beta Function Ratio.* Appl. Statist., 39:2, 309-310. DOI: https://doi.org/10.2307/2347779

[Berry_etal:1991] Berry, K.J., Mielke, P.W., and Cran, G.W., (1991) *Correction to Algorithm AS R83: A remark on Algorithm AS 109: Inverse of the Incomplete Beta Function Ratio.* Appl. Statist., 40:1, p.236.

[Berry_etal:2005] Berry, M.W., Pulatova, S.A., and Stewart, G.W., (2005) *Algorithm 844: Computing sparse reduced-rank approximations to sparse matrices.* ACM Transactions on Mathematical Software, 31:2, 252-269. DOI: https://doi.org/10.1145/1067967.1067972

[Best_Roberts:1975] Best, D.J., and Roberts, D.E., (1975) *Algorithm AS 91: The Percentage Points of the chi2 Distribution.* Appl. Statist., 24:3, 385-388. DOI: https://doi.org/10.2307/2347113

[Bini_etal:2005] Bini, D.A., Gemignani, L., and Tisseur, F., (2005) *The Ehrlich-Aberth method for the nonsymmetric tridiagonal eigenvalue problem.* SIAM J. Matrix Anal. Appl., 27:1, 153-175. DOI: https://doi.org/10.1137/S0895479803429788

[Bjornsson_Venegas:1997] Bjornsson, H., and Venegas, S.A., (1997) *A manual for EOF and SVD analyses of climate data.* McGill University, CCGCR Report No. 97-1, Montreal, Quebec, 52 PP. See: https://www.jsg.utexas.edu/fu/files/EOFSVD.pdf

[blas1] Lawson, C.L., Hanson, R.J., Kincaid, D.R., and Krogh, F.T., (1979) *Algorithm 539: Basic linear algebraic subprograms for fortran usage.* ACM Trans. Math. Software, 5:3, 324-325. DOI: http://dx.doi.org/10.1145/355841.355848

[blas2] Dongarra, J.J., Du Croz, J., Hammarling, S., and Hanson, R.J., (1988) *Algorithm 656: An extended set of basic linear algebra subprograms: Model implementation and test programs.* ACM Trans. Math. Software, 14:1, 18-32. DOI: http://dx.doi.org/10.1145/42288.42292

[blas3] Dongarra, J.J., Du Croz, J., Hammarling, S., and Duff, I., (1990) *Algorithm 679: A set of level 3 basic linear algebra subprograms.* ACM Trans. Math. Software, 16:1, 18-28. DOI: http://dx.doi.org/10.1145/77626.77627

[Bloomfield:1976] Bloomfield, P., (1976) *Fourier analysis of time series- An introduction.* John Wiley and Sons, New York. ISBN: 978-0-471-65399-8

[Blue:1978] Blue, J.L., (1978) *A portable Fortran program to find the Euclidean norm of a vector.* ACM Trans. Math. Soft., 4:1, 15-23. DOI: https://doi.org/10.1145/355769.355771

[Bosner_Barlow:2007] Bosner, N., and Barlow, J.L., (2007) *Block and Parallel versions of one-sided bidiagonalization.* SIAM J. Matrix Anal. Appl., 29:3, 927-953. DOI: https://doi.org/10.1137/050636723

[Braun_Kulperger:1997] Braun, W.J., and Kulperger, R.J., (1997) *Properties of a fourier bootstrap method for time series.* Communications in Statistics - Theory and Methods, 26, 1329-1336. DOI: http://dx.doi.org/10.1080/03610929708831985

[Bretherton_etal:1992] Bretherton, C., Smith, c., and Wallace, J.M., (1992) *An intercomparison of methods for finding coupled patterns in climate data.* Journal of Climate, 5, 541-560. DOI: 10.1175/1520-0442(1992)005<0541:AIOMFF>2.0.CO;2

[Buckley:1994a] Buckley, A.G., (1994) *Conversion to Fortran 90: a Case Study.* ACM Transactions on Mathematical Software, 20(3), 308-353. DOI: http://dx.doi.org/10.1145/192115.192139

[Buckley:1994b] Buckley, A.G., (1994) *Algorithm 734: A Fortran 90 Code for Unconstrained Nonlinear Minimization.* ACM Transactions on Mathematical Software*, 20(3), 354-372. DOI: http://dx.doi.org/10.1145/192115.192146

[Clarkson_Jennrich:1988] Clarkson, D.B., and Jennrich, R.I., (1988) *Quartic rotation criteria and algorithms.* Psychometrika, 53, 251-259. DOI: https://doi.org/10.1007/BF02294136

[Cleveland:1979] Cleveland, W.S., (1979) *Robust Locally Weighted Regression and Smoothing Scatterplots.* Journal of the American Statistical Association, 74, 829-836. DOI: https://doi.org/10.1080/01621459.1979.10481038

[Cleveland_Devlin:1988] Cleveland, W.S., and Devlin, S.J., (1988) *Locally Weighted Regression: An Approach to Regression Analysis by Local Fitting.* Journal of the American Statistical Association, 83, 596-610. DOI: https://doi.org/10.1080/01621459.1988.10478639

[Cleveland_etal:1990] Cleveland, R.B., Cleveland, W.S., McRae, J.E., and Terpennings, I., (1990) *A Seasonal-Trend Decomposition Procedure Based on Loess.* See http://www.jos.nu/Articles/abstract.asp?article=613

[Coates_Diggle:1986] Coates, D.S., and Diggle, P.J., (1986) *Tests for comparing two estimated spectral densities.* Journal of Time series Analysis, 7:1, 7-20. DOI: https://doi.org/10.1111/j.1467-9892.1986.tb00482.x

[Cody:1988] Cody, W.J., (1988) *Algorithm 665: MACHAR: A Subroutine to Dynamically Determine Machine Parameters.* ACM Transactions on Mathematical Software, 14:4, 303-311. DOI: https://doi.org/10.1145/50063.51907

[Cody_Coonen:1993] Cody, W.J., and Coonen, J.T., (1993) *Algorithm 722_ Functions to Support the IEEE Standard for Binary Floating-Point Arithmetic.* ACM Transactions on Mathematical Software, 19:4, 443-451. DOI: https://doi.org/10.1145/168173.168185

[Cooley_etal:1969] Cooley, J.W., Lewis, P., and Welch, P., (1969) *The Fast Fourier Transform and its Applications.* IEEE Trans on Education, 12:1, 27-34. DOI: https://doi.org/10.1109/TE.1969.4320436

[Cooley_etal:1970] Cooley, J.W., Lewis, P., and Welch, P., (1970) *The application of the Fast Fourier Transform algorithm to the estimation of spectra and cross-spectra.* Journal of Sound and Vibration, 12:3, 339-352. DOI: https://doi.org/10.1016/0022-460X(70)90076-3

[Cooper:1968] Cooper, B.E., (1968) *The integral of student's t distribution (Algorithm AS3).* Applied Statistics, 17:2, 189-190. DOI: https://doi.org/10.2307/2985684

[Cran_etal:1977] Cran, G.W., Martin, K.J., and Thomas, G.E., (1977) *Remark AS R19 and Algorithm AS 109: A remark on Algorithms: AS 63 the Incomplete Beta integral, AS 64 Inverse of the Incomplete Beta Function Ratio.* Appl. Statist., 26:1, 111-114. DOI: https://doi.org/10.2307/2346887

[Davison_Hinkley:1997] Davison, A.C., and Hinkley, D.V., (1997) *Bootstrap methods and their application.* Cambridge Univesity press, Cambridge, UK, 484 pp. DOI: https://doi.org/10.1017/CBO9780511802843

[Demmel_Kahan:1990] Demmel, J.W., and Kahan, W., (1990) *Accurate singular values of bidiagonal matrices.* SIAM Journal of Scientific and Statistical Computing, 11:5, 873-912. DOI: https://doi.org/10.1137/0911052

[Demmel_etal:1993] Demmel, J., Heath, M.T., and Van Der Vorst, H., (1993) *Parallel numerical linear algebra.* Acta Numerica, 2, 111-197. DOI: https://doi.org/10.1017/S096249290000235X

[Dhillon:1998] Dhillon, I.S., (1998) *Current inverse iteration software can fail.* BIT, 38:4, 685-704. DOI: https://doi.org/10.1007/BF02510409

[Diggle:1990] Diggle, P.J., (1990) *Time series: a biostatistical introduction.* Clarendon Press, Oxford, 257 pp. ISBN-10: 0198522266

[Diggle_Fisher:1991] Diggle, P.J., and Fisher, N.I., (1991) *Nonparametric comparison of cumulative periodograms.* Applied Statistics, 40:3, 423-434. DOI: https://doi.org/10.2307/2347522

[Dongarra_etal:1989] Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., (1989) *Block reduction of matrices to condensed form for eigenvalue computations.* Journal of Computational and Applied Mathematics, 27:1-2, 215-227. DOI: https://doi.org/10.1016/B978-0-444-88621-7.50015-3

[Dongarra_etal:2018] Dongarra, J.J., Gates, M., Haidar, A., Kurzak, J., Luszczek, P., Tomov, S., Yamazaki, I., (2018) *The Singular Value Decomposition: Anatomy of Optimizing an Algorithm for Extreme Scale.* SIAM Review, 60:4, 808-865. DOI: https://doi.org/10.1137/17M1117732

[Dongarra_etal:2019] Dongarra, J.J., Gates, M., Haidar, A., Kurzak, J., Luszczek, P., Wu, P., Yamazaki, I., Yarkhan, A., Abalenkovs, M., Bagherpour, N., Hammarling, S., Sistek, J., Stevens, D., Zounon, M., Relton, S.D., (2019) *PLASMA: Parallel Linear Algebra Software for Multicore Using OpenMP.* ACM Trans. Math. Softw., 45:2, Article 16 (April 2019), 35 pages. DOI: https://doi.org/10.1145/3264491

[Doornik:2007] Doornik, J.A, (2007) *Conversion of high-period random numbers to floating point.* ACM Transactions on Modeling and Computer Simulation, 17:1, Article No. 3. DOI: https://doi.org/10.1145/1189756.1189759

[Duchon:1979] Duchon, C., (1979) *Lanczos filtering in one and two dimensions.* Journal of applied meteorology, 18:8, 1016-1022. DOI: 10.1175/1520-0450(1979)018<1016:LFIOAT>2.0.CO;2

[Duersch_Gu:2017] Duersch, J.A., and Gu, M., (2017) *Randomized QR with column pivoting.* SIAM J. Sci. Comput., 39:4, C263-C291. DOI: https://doi.org/10.1137/15M1044680

[Duersch_Gu:2020] Duersch, J.A., and Gu, M., (2020) *Randomized projection for rank-revealing matrix factorizations.* SIAM Review, 62:3, 661-682. DOI: http://doi.org/10.1137/20m1335571

[Ebisuzaki:1997] Ebisuzaki, W., (1997) *A method to estimate the statistical significance of a correlation when the data are serially correlated.* Journal of climate, 10, 2147-2153. DOI: https://doi.org/10.1175/1520-0442%281997%29010%3C2147%3AAMTETS%3E2.0.CO%3B2

[Erichson_etal:2019] Erichson, N.B., Voronin, S., Brunton, S.L., and Kutz, J.N., (2019) *Randomized matrix decompositions using R.* arXiv.1608.02148. See https://arxiv.org/abs/1608.02148

[Feng_etal:2019] Feng, Y., Xiao, J., and Gu, M., (2019) *Flip-flop spectrum-revealing QR factorizations and its applications to singular value decomposition.* Electronic Transactions on Numerical Analysis (ETNA), 51, 469-494. DOI: https://doi.org/10.1553/etna_vol51s469

[Fernando:1997] Fernando, K.V., (1997) *On computing an eigenvector of a tridiagonal matrix. Part I: Basic results.* Siam J. Matrix Anal. Appl., 18:4, 1013-1034. DOI: https://doi.org/10.1137/S0895479895294484

[Fernando:1998] Fernando, K.V., (1998) *Accurately counting singular values of bidiagonal matrices and eigenvalues of skew-symmetric tridiagonal matrices.* SIAM J. Matrix Anal. Appl., 20:2, 373-399. DOI: https://doi.org/10.1137/S089547989631175X

[Fernando_Parlett:1994] Fernando, K.V., and Parlett, B.N., (1994) *Accurate singular values and differential qd algorithms.* Numer. Math., 67:2, 191-229. DOI: https://doi.org/10.1007/s002110050024

[Fortran] Metcalf, M., Reid, J., and Cohen, M., (2013) *Modern FORTRAN Explained.* 7rd Ed., Oxford University Press, Oxford, UK.

[Gentleman_Marovich:1974] Gentleman, W.M., and Marovich, S.B., (1974) *More on algorithms that reveal properties of floating point arithmetic units.* Communications of the ACM, 17:5, 276-277. DOI: https://doi.org/10.1145/360980.361003

[Godunov_etal:1993] S.K. Godunov, A.G. Antonov, O.P. Kiriljuk, V.I. Kostin (1993) *Guaranteed Accuracy in Numerical Linear Algebra.* Kluwer Academic. (A revised translation of a Russian text first published in 1988 in Novosibirsk)

[Goertzel:1958] Goertzel, G., (1958) *An Algorithm for the Evaluation of Finite Trigonometric Series.* The American Mathematical Monthly, 65:1, 34-35. DOI: https://doi.org/10.2307/2310304

[Goldstein:1973] Goldstein, R.B., (1973) *Chi-square quantiles.* Comm. A.C.M., 16:8, 483-485. DOI: https://doi.org/10.1145/355609.362319

[Golub_VanLoan:1996] Golub, G.H., and Van Loan, C., (1996) *Matrix Computations.* 3rd Ed., The John Hopkins University Press, Baltimore, MD.

[Greenbaum_Dongarra:1989] Greenbaum, A., and Dongarra, J., (1989) *Experiments with QR/QL Methods for the Symmetric Tridiagonal Eigenproblem.* LAPACK Working Note No 17.

[Gu:2015] Gu, M., (2015) *Subspace iteration randomization and singular value problems.* SIAM J. Sci. Comput. Comm., 37, A1139-A1173. DOI: https://doi.org/10.1137/130938700

[Halko_etal:2011] Halko, N., Martinsson, P.G., Tropp, J.A., (2011) *Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions.* SIAM Rev., 53, 217-288. DOI: https://doi.org/10.1137/090771806

[Hansen_etal:2012] Hansen, P.C., Pereyra, V., and Scherer, G., (2012) *Least Squares Data Fitting with Applications.* Johns Hopkins University Press, 328 pp. ISBN:9781421407869

[Hanson_Hopkins:2018] Hanson, R.J., and Hopkins, T., (2018) *Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm.* ACM Trans. Math. Soft., 44:3, Article 24, 1-23. DOI: https://doi.org/10.1145/3134441

[Harase:2014] Harase, S., (2014) *On the F2-linear relations of Mersenne Twister pseudorandom number generators.* Mathematics and Computers in Simulation, 100, 103-113. DOI: https://doi.org/10.1016/j.matcom.2014.02.002

[Hart:1978] Hart, J.F., (1978) *Computer Approximations.* Krieger Publishing Co., Inc. Melbourne, FL, USA. ISBN:0882756427

[Hegland_etal:1999] Hegland, M., Kahn, M., and Osborn, M., (1999) *A parallel algorithm for the reduction to tridiagonal form for eigendecomposition.* SIAM Journal on Scientific Computing, 21:3, 987-1005. DOI: https://doi.org/10.1137/S1064827595296719

[Hennecke:1995] Hennecke, M., (1995) *A Fortran90 interface to random number generation.* Computer programs in physics, 90 (1), 117-120. DOI: https://doi.org/10.1016/0010-4655(95)00065-N

[Higham:2009] Higham, N.J., (2009) *Cholesky factorization.* Wiley Interdisciplinary Reviews: Computational Statistics, 1, 251-254. DOI: https://doi.org/10.1002/wics.018

[Higham:2011] Higham, N.J., (2011) *Gaussian elimination.* Wiley Interdisciplinary Reviews: Computational Statistics, 3:3, 230-238. DOI: https://doi.org/10.1002/wics.164

[Hill:1970] Hill, G.W., (1970) *Student's t-distribution (Algorithm 395).* Comm. A.C.M., 13:10, 617-619. DOI: https://doi.org/10.1145/355598.355599

[Hill:1970b] Hill, G.W., (1970) *Student's t-quantiles (Algorithm 396).* Comm. A.C.M., 13:10, 619-620. DOI: https://doi.org/10.1145/355598.355600

[Hill:1973] Hill, I.D., (1973) *Algorithm AS66: The Normal Integral.* Appl. Statist., 22:3, 424-427. DOI: https://doi.org/10.2307/2346800

[Howell_etal:2008] Howell, G.W., Demmel, J., Fulton, C.T., Hammarling, S., and Marmol, K., (2008) *Cache efficient bidiagonalization using BLAS 2.5 operators.* ACM Transactions on Mathematical Software (TOMS), 34:3, Article 14. DOI: https://doi.org/10.1145/1356052.1356055

[Huckaby_Chan:2003] Huckaby, D.A., and Chan, T.F., (2003) *On the convergence of Stewart's QLP algorithm for approximating the SVD.* Numer. Algorithms, 32, 287-316. DOI: https://doi.org/10.1023/A:1024082314087

[Huckaby_Chan:2005] Huckaby, D.A., and Chan, T.F., (2003) *Stewart's pivoted QLP decomposition for low-rank matrices.* Numerical Linear Algebra with Applications, 12:2-3, 153-159. DOI: https://doi.org/10.1002/nla.404

[Iacobucci_Noullez:2005] Iacobucci, A., and Noullez, A., (2005) *A Frequency Selective Filter for Short-Length Time Series.* Computational Economics, 25:1-2,75-102. DOI: https://doi.org/10.1007/s10614-005-6276-7

[Ipsen:1997] Ipsen, I.C.F., (1997) *Computing an eigenvector with inverse iteration.* SIAM Review, 39:2, 254-291. DOI: https://doi.org/10.1137/S0036144596300773

[Jackson:2003] Jackson, J.E., (2003) *A user's guide to principal components.* 592 pp., John Wiley and Sons, New York, USA. ISBN 978-0-471-47134-9.

[Jenkins_Watts:1968] Jenkins, G.M., and Watts, D.G., (1968) *Spectral Analysis and its Applications.* San Francisco: Holden-Day. ISBN-10: 0816244642

[Jennrich:1970] Jennrich, R.I., (1970) *Orthogonal rotation algorithms.* Psychometrika, 35, 229-235. DOI: https://doi.org/10.1007/BF02291264

[Jolliffe:2002] Jolliffe, I.T., (2002) *Principal component analysis.* 2nd Ed, 487 pp., Springer-Verlag, New York, USA. ISBN 978-0-387-22440-4.

[Knuth:1997] Knuth, D.E., (1997) *The Art of Computer Programming, Volume III: Sorting and Searching.* 3rd Ed, Addison-Wesley, Reading, MA, USA. ISBN 0201896850.

[Lanczos:1964] Lanczos, C., (1964) *A precision approximation of the gamma function.* J. SIAM Numer. Anal., B, 1:1, 86-96. DOI: https://doi.org/10.1137/0701008

[Lang:1998] Lang, B., (1998) *Using level 3 BLAS in rotation-based algorithms.* Siam J. Sci. Comput., 19:2, 626-634. DOI: https://doi.org/10.1137/S1064827595280211

[Lau:1980] Lau, C.L., (1980) *Algorithm AS 147: A simple series for the Incomplete Gamma Integral.* Appl. Statist., 29:1, 113-114. DOI: https://doi.org/10.2307/2346431

[Lawson_Hanson:1974] Lawson, C.L., and Hanson, R.J., (1974) *Solving least square problems.* Prentice-Hall. DOI: https://doi.org/10.1137/1.9781611971217

[LEcuyer:1999] L'Ecuyer, P., (1999) *Tables of Maximally-Equidistributed Combined LFSR Generators.* Mathematics of computations, 68:225, 261-269. DOI: https://doi.org/10.1090/S0025-5718-99-01039-X

[Li_etal:2014] Li, S., Gu, M., and Parlett, B. N., (2014) *An Improved DQDS Algorithm.* SIAM J. Sci. Comput., 36:3, C290-C308. DOI: https://doi.org/10.1137/120881087

[Li_etal:2017] Li, H.,Linderman, G.C., Szlam, A., Stanton, K.P., Kluger, Y., and Tygert, M., (2017) *Algorithm 971: An implementation of a randomized algorithm for principal component analysis.* ACM Transactions on Mathematical Software (TOMS), 43:3, Article 28. DOI: https://doi.org/10.1145/3004053

[Mahoney_Drineas:2009] Mahoney, M.W., and Drineas, P., (2009) *CUR matrix decompositions for improved data analysis.* PNAS, 106:3, 697-702. DOI: https://doi.org/10.1073/pnas.0803205106

[Majumder_Bhattacharjee:1973] Majumder, K.L., and Bhattacharjee, G.P., (1973) *Algorithm AS 63: the Incomplete Beta Integral.* Appl. Statist., 22:3, 409-411. DOI: https://doi.org/10.2307/2346797

[Malcolm:1972] Malcolm, M.A., (1972) *Algorithms to reveal properties of floating-point arithmetic.* Communications of the ACM, 15:11, 949-951. DOI: https://doi.org/10.1145/355606.361870

[Malyshev:2000] Malyshev, A.N., (2000) *On deflation for symmetric tridiagonal matrices.* Report 182 of the Department of Informatics, University of Bergen, Norway. See: https://www.ii.uib.no/~sasha/mypapers/report/ii182.ps.gz

[Marques_Vasconcelos:2017] Marques, O., and Vasconcelos, P.B., (2017) *Computing the Bidiagonal SVD Through an Associated Tridiagonal Eigenproblem.* In: Dutra I., Camacho R., Barbosa J., Marques O. (eds) High Performance Computing for Computational Science - VECPAR 2016. VECPAR 2016. Lecture Notes in Computer Science, vol 10150. Springer, Cham. DOI: https://doi.org/10.1007/978-3-319-61982-8_8

[Marques_etal:2020] Marques, O., Demmel, J., and Vasconcelos, P.B., (2020) *Bidiagonal SVD Computation via an Associated Tridiagonal Eigenproblem.* ACM Trans. Math. Softw. 46:2, Article 14, 1-25. DOI: https://doi.org/10.1145/3361746

[Marsaglia:1999] Marsaglia, G., (1999) *Random number generators for Fortran.* Posted to the computer-programming-forum. See: http://computer-programming-forum.com/49-fortran/b89977aa62f72ee8.htm

[Marsaglia:2005] Marsaglia, G., (2005) *Double precision RNGs.* Posted to the electronic billboard to sci.math.num-analysis. See: http://sci.tech-archive.net/Archive/sci.math.num-analysis/2005-11/msg00352.html

[Marsaglia:2007] Marsaglia, G., (2007) *Fortran and C: United with a KISS.* Posted to the Google comp.lang.forum. See: http://groups.google.co.uk/group/comp.lang.fortran/msg/6edb8ad6ec5421a5

[Martinsson_Voronin:2016] Martinsson, P.G., and Voronin, S., (2016) *A randomized blocked algorithm for efficiently computing rank-revealing factorizations of matrices.* SIAM J. Sci. Comput., 38:5, S485-S507. DOI: https://doi.org/10.1137/15M1026080

[Martinsson_etal:2017] Martinsson, P.G., Quintana-Orti, G., Heavner, N., and Van de Geijn, R., (2017) *Householder QR factorization with randomization for column pivoting (HQRRP).* SIAM J. Sci. Comput., 39:2, C96-C115. DOI: https://doi.org/10.1137/16M1081270

[Martinsson:2019] Martinsson, P.G., (2019) *Randomized methods for matrix computations.* arXiv.1607.01649. See https://arxiv.org/abs/1607.01649

[Mary_etal:2015] Mary, T., Yamazaki, I., Kurzak, J., Luszczek, P., Tomov, S., and Dongarra, J., (2015) *Performance of Random Sampling for Computing Low-rank Approximations of a Dense Matrix on GPUs.* International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15). DOI: https://doi.org/10.1145/2807591.2807613

[Mastronardi_etal:2006] Mastronardi, M., Van Barel, M., Van Camp, E., and Vandebril, R., (2006) *On computing the eigenvectors of a class of structured matrices.* Journal of Computational and Applied Mathematics, 189:1-2, 580-591. DOI: https://doi.org/10.1016/j.cam.2005.03.048

[Matsumoto_Nishimura:1998] Matsumoto, M., and Nishimura, T., (1998) *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator.* ACM Transactions on Modeling and Computer Simulation, 8:1, 3-30. DOI: https://doi.org/10.1145/272991.272995

[Monro_Branch:1977] Monro, D.M., and Branch, J.L., (1997) *Algorithm AS 117: The Chirp discrete Fourier transform of general length.* Appl. Statist., 26:3, 351-361. DOI: https://doi.org/10.2307/2346986

[Musco_Musco:2015] Musco, C., and Musco, C., (2015) *Randomized block krylov methods for stronger and faster approximate singular value decomposition.* In Proceedings of the 28th International Conference on Neural Information Processing Systems, NIPS 15, pages 1396-1404, Cambridge, MA, USA, 2015. MIT Press.

[Nakatsukasa_etal:2012] Nakatsukasa, Y., Aishima, K., and Yamazaki, I., (2012) *dqds with aggressive early deflation.* SIAM Journal on Matrix Analysis Applications, 33:1, 22-51. DOI: https://doi.org/10.1137/110821330

[Noreen:1989] Noreen, E.W., (1989) *Computer-intensive methods for testing hypotheses: an introduction.* Wiley and Sons, New York, USA, ISBN:978-0-471-61136-3

[Olagnon:1996] Olagnon, M., (1996) *Traitement de donnees numeriques avec Fortran 90.* Masson, 264 pages, Chapter 11.1.2, ISBN 2-225-85259-6. (in French)

[Oppenheim_Schafer:1999] Oppenheim, A.V., and Schafer, R.W., (1999) *Discrete-Time Signal Processing.* 2rd Edition. Prentice-Hall, Signal Processing Series, New Jersey. ISBN-10: 0131988425

[Parlett_Dhillon:1997] Parlett, B.N., and Dhillon, I.S., (1997) *Fernando's solution to Wilkinsin's problem: An application of double factorization.* Linear Algebra and its Applications, 267, 247-279. DOI: https://doi.org/10.1016/S0024-3795(97)80053-5

[Parlett:1998] Parlett, B.N., (1998) *The Symmetric Eigenvalue Problem.* Revised edition, SIAM, Philadelphia. DOI: https://doi.org/10.1137/1.9781611971163

[Parlett_Marques:2000] Parlett, B.N., and Marques, O.A., (2000) *An implementation of the dqds algorithm (positive case).* Linear Algebra and its Applications, 309, 217-259. DOI: https://doi.org/10.1016/S0024-3795(00)00010-0

[Peizer_Pratt:1968] Peizer, D.B., and Pratt, J.W., (1968) *A normal approximation for Binomial, F, Beta, and other common, related tail probabilities, I.* J.A.S.A., 63:324, 1457-1483. DOI: https://doi.org/10.2307/2285895

[Potscher_Reschenhofer:1988] Potscher, B.,M., and Reschenhofer, E., (1988) *Discriminating between two spectral densities in case of replicated observations.* Journal of Time series Analysis, 9:3, 221-224. DOI: https://doi.org/10.1111/j.1467-9892.1988.tb00466.x

[Potscher_Reschenhofer:1989] Potscher, B.,M., and Reschenhofer, E., (1989) *Distribution of the Coates-Diggle test statistic in case of replicated observations.* Statistics, 20:3, 417-421. DOI: https://doi.org/10.1080/02331888908802190

[Priestley:1981] Priestley, M.B., (1981) *Spectral Analysis and Time Series.* London: Academic Press. ISBN-10: 0125649223

[Ralha:2003] Ralha, R.M.S., (2003) *One-sided reduction to bidiagonal form.* Linear Algebra Appl., 358:1-3, 219-238. DOI: https://doi.org/10.1016/S0024-3795(01)00569-9

[Reinsch_Bauer:1968] Reinsch, C., and Bauer, F.L., (1968) *Rational QR transformation with Newton shift for symmetric tridiagonal matrices.* Numerische Mathematik, 11, 264-272. DOI: https://doi.org/10.1007/978-3-662-39778-7_17

[Reinsch_Richter:2023] Reinsch, C., and Richter, M., (12023) *Singular value decomposition in extended double precision arithmetic.* Numerical Algorithms, 93,1137-1155. DOI: https://doi.org/10.1007/s11075-022-01459-9

[Sedgewick:1998] Sedgewick, R., (1998) *Algorithms in C - Parts 1-4: Fundamentals, Data Structures, Sorting, Searching.* 3rd Ed, Addison-Wesley, Reading, MA, USA. ISBN 978-0-201-31452-6.

[Shea:1988] Shea, B.L., (1988) *Algorithm AS 239: Chi-squared and Incomplete Gamma Integral.* Appl. Statist., 37:3, 466-473. DOI: https://doi.org/10.2307/2347328

[Shea:1991] Shea, B.L., (1991) *Algorithm AS R85 : A remark on AS 91: The Percentage Points of the chi2 Distribution.* Appl. Statist., 40:1, pp.233-235. DOI: https://doi.org/10.2307/2347937

[Stewart:1980] Stewart, G.W., (1980) *The efficient generation of random orthogonal matrices with an application to condition estimators.* SIAM J. Numer. Anal., 17:3, 403-409. DOI: https://doi.org/10.1137/0717034

[Stewart:1999] Stewart, G.W., (1999) *Four algorithms for the the efficient computation of truncated pivoted qr approximations to a sparse matrix.* Numerische Mathematik, 83, 313-323. DOI: https://doi.org/10.1007/s002110050451

[Stewart:1999b] Stewart, G.W., (1999) *The QLP approximation to the singular value decomposition.* SIAM J. Sci. Comput., 20:4, 1336-1348. DOI: https://doi.org/10.1137/S1064827597319519

[Stewart:2007] Stewart, G.W., (2007) *Block Gram-Schmidt Orthogonalization.* Report TR-4823, Department of Computer Science, College Park, University of Maryland.

[Terray_etal:2003] Terray, P., Delecluse, P., Labattu, S., Terray, L., (2003) *Sea Surface Temperature associations with the Late Indian Summer Monsoon.* Climate Dynamics, 21:7-8, 593-618. DOI: https://doi.org/10.1007/s00382-003-0354-0

[Theiler_etal:1992] Theiler, J., Eubank, S., Longtin, A., Galdrikian, B., and Farmer, J.D. (1992) *Testing for nonlinearity in time series: the method of surrogate data.* Physica D, 8, 77-94. DOI: https://doi.org/10.1016/0167-2789(92)90102-s

[Thomas_etal:2007] Thomas, D.B., Luk, W., Leong, P.H.W., and Villasenor, J.D., (2007) *Gaussian random number generators.* ACM Comput. Surv., 39:4, Article 11, 38 pages. DOI: https://doi.org/10.1145/1287620.1287622. See: http://doi.acm.org/10.1145/1287620.1287622

[Tropp_Webber:2023] Tropp, J.A., and Webber, R.J.,(2023) *Randomized algorithms for low-rank matrix approximation: Design, analysis, and applications.* arXiv:2306.12418. See: https://arxiv.org/abs/2306.12418

[VanZee_etal:2011] Van Zee, F.G., Van de Geijn, R., and Quintana-Orti, G., (2011) *Restructuring the QR Algorithm for High-Performance Application of Givens Rotations.* FLAME Working Note 60. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-11-36.

[vonStorch_Zwiers:2002] von Storch, H., and Zwiers, F.W., (2002) *Statistical Analysis in Climate Research.* Cambridge University Press, Cambridge, UK, 484 pp., ISBN:9780521012300

[Voronin_Martinsson:2015] Voronin, S., and Martinsson, P.G., (2015) *Rsvdpack: Subroutines for computing partial singular value decompositions via randomized sampling on single core, multi core, and gpu architectures.* arXiv.1502.05366. See https://arxiv.org/abs/1502.05366

[Voronin_Martinsson:2017] Voronin, S., and Martinsson, P.G., (2017) *Efficient algorithms for cur and interpolative matrix decompositions.* Adv. Comput. Math., 43, 495-516. DOI: https://doi.org/10.1007/s10444-016-9494-8

[Walck:2007] Walck, C., (2007) *Hand-book on statistical distributions for experimentalists.* Stockholm University, Internal Report SUF-PFY/96-01. See http://staff.fysik.su.se/~walck/suf9601.pdf

[Walker:1988] Walker, H.F., (1988) *Implementation of the GMRES method using Householder transformations.* Siam J. Sci. Stat. Comput., 9:1, 152-163. DOI: https://doi.org/10.1137/0909010

[Welch:1967] Welch, P.D., (1967) *The use of Fast Fourier Transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms.* IEEE trans. on audio and electroacoustics, 15:2, 70-73. DOI: https://doi.org/10.1109/TAU.1967.1161901

[Wichura:1988] Wichura, M.J., (1988) *Algorithm AS 241: The percentage points of the normal distribution.* Appl. Statist., 37:3, 477-484. DOI: https://doi.org/10.2307/2347330

[Wills_etal:2018] Wills, R.C., Schneider, T., Wallace, J.M., Battisti, D.S., and Hartmann, D.L., (2018) *Disentangling Global Warming, Multidecadal Variability, and El Nino in Pacific Temperatures.* Geophysical Research Letters, 45:5, 2487-2496. DOI: https://doi.org/10.1002/2017GL076327

[Wilson_Hilferty:1931] Wilson, E.B., and Hilferty, M.M., (1931) *The distribution of Chi-square.* Proc. Natl. Acad. Sci., USA, 17:12, 684-688. DOI: https://doi.org/10.1073/pnas.17.12.684

[Wu_Xiang:2020] Wu, N., and Xiang, H., (2020) *Randomized QLP decomposition.* Linear algebra and its applications, 599:15, 18-35. DOI: https://doi.org/10.1016/j.laa.2020.03.041

[Xiao_etal:2017] Xiao, J., Gu, M., and Langou, J., (2017) *Fast parallel randomized QR with column pivoting algorithms for reliable low-rank matrix approximations.* IEEE 24th International Conference on High Performance Computing (HiPC), IEEE, 2017, 233-242. See https://ieeexplore.ieee.org/document/8287754

[Yamashita_etal:2013] Yamashita, T., Kimura, K., Takata, M., and Nakamura, Y., (2013) *An application of the Kato-Temple inequality on matrix eigenvalues to the dqds algorithm for singular values.* JSIAM Letters, 5, 21-24. DOI: https://doi.org/10.14495/jsiaml.5.21

[YarKhan_etal:2016] YarKhan, A., Kurzak, J., Luszczek, P., Dongarra J.J., (2016) *Porting the PLASMA numerical library to the OpenMP standard.* Int. J. Parallel Program, 45:3, 1-22. DOI: https://doi.org/10.1007/s10766-016-0441-6

[Yu_etal:2018] Yu, W., Gu, Y., and Li, Y., (2018) *Efficient randomized algorithms for the fixed-precision low-rank matrix approximation.* SIAM J. Mat. Ana. Appl., 39:3, 1339-1359. DOI: https://doi.org/10.1137/17M1141977

INDEX

A

abse() (*built-in function*), 68
apply_h1() (*built-in function*), 100
apply_h2() (*built-in function*), 101
apply_hous1() (*built-in function*), 98
apply_hous2() (*built-in function*), 99
apply_p_bd() (*built-in function*), 144
apply_q_bd() (*built-in function*), 143
apply_q_lq() (*built-in function*), 103
apply_q_qr() (*built-in function*), 109
apply_q_qr_ts() (*built-in function*), 109
apply_q_symtrid() (*built-in function*), 114
apply_rot_fastgivens() (*built-in function*), 93
apply_rot_givens() (*built-in function*), 91
ARCH, 9
ARCHFLAGS, 9
array_copy() (*built-in function*), 64
arth() (*built-in function*), 66
asc2ebc() (*built-in function*), 55
ascii_case_change() (*built-in function*), 56
ascii_is_alpha() (*built-in function*), 54
ascii_is_digit() (*built-in function*), 54
ascii_is_lower() (*built-in function*), 54
ascii_is_same() (*built-in function*), 54
ascii_is_upper() (*built-in function*), 54
ascii_string_comp() (*built-in function*), 55
ascii_string_eq() (*built-in function*), 55
ascii_string_index() (*built-in function*), 55
ascii_to_lower() (*built-in function*), 55
ascii_to_upper() (*built-in function*), 55
assert() (*built-in function*), 65
assert_eq() (*built-in function*), 66
axpy() (*built-in function*), 266

B

bd_cmp() (*built-in function*), 139
bd_cmp2() (*built-in function*), 140
bd_cmp3() (*built-in function*), 141
bd_coef() (*built-in function*), 250
bd_coef2() (*built-in function*), 251
bd_deflate() (*built-in function*), 179
bd_deflate2() (*built-in function*), 180
bd_dqds() (*built-in function*), 147
bd_inviter() (*built-in function*), 173
bd_inviter2() (*built-in function*), 174
bd_lasq1() (*built-in function*), 147
bd_max_singval() (*built-in function*), 146
bd_singval() (*built-in function*), 145
bd_singval2() (*built-in function*), 146
bd_svd() (*built-in function*), 144
bd_svd2() (*built-in function*), 145
bdsdc() (*built-in function*), 271
bdsqr() (*built-in function*), 271
bdsvdx() (*built-in function*), 271
bootstrap_cor() (*built-in function*), 225

C

case_change() (*built-in function*), 56
center() (*built-in function*), 56
CHECKFLAGS, 14, 23
chol_cmp() (*built-in function*), 191
chol_cmp2() (*built-in function*), 191
chol_solve() (*built-in function*), 194
comp_anoma() (*built-in function*), 220
comp_anoma_grp() (*built-in function*), 221
comp_anoma_grp_miss() (*built-in function*), 221
comp_anoma_miss() (*built-in function*), 220
comp_composite() (*built-in function*), 221
comp_composite_miss() (*built-in function*), 222
comp_conflim() (*built-in function*), 255
comp_cor() (*built-in function*), 223
comp_cor_miss() (*built-in function*), 223
comp_cor_miss2() (*built-in function*), 224
comp_cormat() (*built-in function*), 227
comp_cormat_miss() (*built-in function*), 227
comp_det() (*built-in function*), 197
comp_eof() (*built-in function*), 227
comp_eof2() (*built-in function*), 228
comp_eof3() (*built-in function*), 228
comp_eof_miss() (*built-in function*), 229
comp_eof_miss2() (*built-in function*), 229
comp_eof_miss3() (*built-in function*), 230
comp_filt_rot_pc() (*built-in function*), 232
comp_ginv() (*built-in function*), 182

comp_inv() (*built-in function*), 195
comp_lfc_rot_pc() (*built-in function*), 231
comp_mca() (*built-in function*), 232
comp_mca2() (*built-in function*), 233
comp_mca_miss() (*built-in function*), 233
comp_mca_miss2() (*built-in function*), 234
comp_mvs() (*built-in function*), 216
comp_mvs_grp() (*built-in function*), 218
comp_mvs_grp_miss() (*built-in function*), 218
comp_mvs_miss() (*built-in function*), 217
comp_ortho_rot_eof() (*built-in function*), 231
comp_pc() (*built-in function*), 235
comp_pc_eof() (*built-in function*), 230
comp_pc_mca() (*built-in function*), 235
comp_pc_miss() (*built-in function*), 235
comp_smooth() (*built-in function*), 242
comp_smooth_rot_pc() (*built-in function*), 231
comp_stl() (*built-in function*), 244
comp_stlez() (*built-in function*), 243
comp_sym_ginv() (*built-in function*), 196
comp_sym_inv() (*built-in function*), 196
comp_trend() (*built-in function*), 242
comp_triang_inv() (*built-in function*), 196
comp_unistat() (*built-in function*), 215
comp_unistat_miss() (*built-in function*), 216
comp_uut_ltl() (*built-in function*), 197
copy() (*built-in function*), 266
cpusecs() (*built-in function*), 60
cross_spctrm() (*built-in function*), 260
cross_spctrm2() (*built-in function*), 261
cross_spectrum() (*built-in function*), 263
cross_spectrum2() (*built-in function*), 264
cumprod() (*built-in function*), 66
cumsum() (*built-in function*), 66
cur_cmp() (*built-in function*), 88

D

dan_filter() (*built-in function*), 254
data_window() (*built-in function*), 255
day_of_week() (*built-in function*), 58
daynum() (*built-in function*), 57
daynum_to_dayweek() (*built-in function*), 59
daynum_to_ymd() (*built-in function*), 58
define_rot_fastgivens() (*built-in function*), 93
define_rot_fastgivens2() (*built-in function*), 96
define_rot_givens() (*built-in function*), 90
det() (*built-in function*), 197
detrend() (*built-in function*), 245
dflapp() (*built-in function*), 126
dflapp_bd() (*built-in function*), 178
dflgen() (*built-in function*), 125
dflgen2() (*built-in function*), 126
dflgen2_bd() (*built-in function*), 178
dflgen_bd() (*built-in function*), 177
diagadd() (*built-in function*), 71
diagmult() (*built-in function*), 71
DIRLIB, 9, 13, 14, 22
do_index() (*built-in function*), 49
dot() (*built-in function*), 266
dot_product2() (*built-in function*), 62
dotu() (*built-in function*), 266
drawbootsample() (*built-in function*), 89
drawsample() (*built-in function*), 89
DRVFLAGS, 9

E

ebc2asc() (*built-in function*), 55
eig_abs_sort() (*built-in function*), 121
eig_cmp() (*built-in function*), 117
eig_cmp2() (*built-in function*), 118
eig_cmp3() (*built-in function*), 119
eig_cmp4() (*built-in function*), 119
eig_cmp5() (*built-in function*), 119
eig_sort() (*built-in function*), 121
eigval_abs_sort() (*built-in function*), 125
eigval_cmp() (*built-in function*), 122
eigval_cmp2() (*built-in function*), 122
eigval_cmp3() (*built-in function*), 123
eigval_sort() (*built-in function*), 125
eigvalues() (*built-in function*), 121
elapsed_time() (*built-in function*), 59
end_fft() (*built-in function*), 241
enter_proc() (*built-in function*), 50
entering() (*built-in function*), 51
environment variable
    STATPACKDIR, 8
environment variable
    ARCH, 9
    ARCHFLAGS, 9
    CHECKFLAGS, 14, 23
    DIRLIB, 9, 13, 14, 22
    DRVFLAGS, 9
    FORTRAN, 9
    GOTOBLAS_NUM_THREADS, 33
    INTERFACES, 9, 13
    LBLAS, 9, 10
    LD_LIBRARY_PATH, 30
    LIB, 9, 13, 14, 22
    LIBTOOL, 9
    LIBTOOLFLAGS, 9
    LLAPACK, 9, 10
    LOADFLAGS, 9–11, 30
    MKL_NUM_THREADS, 33
    NOOPTFLAGS, 9
    OMP_DYNAMIC, 5, 32
    OMP_MAX_ACTIVE_LEVELS, 5, 32
    OMP_NESTED, 5, 32

OMP_NUM_THREADS, 5, 32
OMP_STACKSIZE, 5, 32
OPENBLAS_NUM_THREADS, 33
OPTFLAGS, 9
OPTS, 9, 10, 13, 16
STATPACKDIR, 8
estim_dof() (*built-in function*), 255
estim_dof2() (*built-in function*), 255
extend() (*built-in function*), 254

F

fastgivens2_vec() (*built-in function*), 96
fastgivens_mat_left() (*built-in function*), 95
fastgivens_mat_right() (*built-in function*), 95
fastgivens_vec() (*built-in function*), 94
fastgivens_vec_mat_left() (*built-in function*), 95
fastgivens_vec_mat_right() (*built-in function*), 95
fft() (*built-in function*), 239
fft_row() (*built-in function*), 240
fftxy() (*built-in function*), 239
find_field() (*built-in function*), 56
FORTRAN, 9
freq_func() (*built-in function*), 252

G

gchol_cmp() (*built-in function*), 192
gchol_cmp2() (*built-in function*), 192
gebrd() (*built-in function*), 271
geev() (*built-in function*), 270
geevx() (*built-in function*), 270
gels() (*built-in function*), 272
gelsd() (*built-in function*), 272
gelss() (*built-in function*), 272
gelsy() (*built-in function*), 272
gemm() (*built-in function*), 267
gemv() (*built-in function*), 266
gen_bd_mat() (*built-in function*), 182
gen_random_mat() (*built-in function*), 81
gen_random_ortho_mat() (*built-in function*), 82
gen_random_sym_mat() (*built-in function*), 81
gen_symtrid_mat() (*built-in function*), 131
geop() (*built-in function*), 66
geqp3() (*built-in function*), 272
geqr2() (*built-in function*), 272
geqrf() (*built-in function*), 272
ger() (*built-in function*), 266
geru() (*built-in function*), 266
gesdd() (*built-in function*), 271
gesv() (*built-in function*), 271
gesvd() (*built-in function*), 271
gesvdx() (*built-in function*), 271
get_date() (*built-in function*), 60
get_date_time() (*built-in function*), 60
get_diag() (*built-in function*), 71
ginv() (*built-in function*), 182
givens_mat_left() (*built-in function*), 92
givens_mat_right() (*built-in function*), 92
givens_vec() (*built-in function*), 92
givens_vec_mat_left() (*built-in function*), 92
givens_vec_mat_right() (*built-in function*), 93
gk_qr_cmp() (*built-in function*), 173
GOTOBLAS_NUM_THREADS, 33

H

h1() (*built-in function*), 100
h2() (*built-in function*), 100
hemm() (*built-in function*), 267
her2() (*built-in function*), 266
her2k() (*built-in function*), 267
herk() (*built-in function*), 267
hous1() (*built-in function*), 98
hous2() (*built-in function*), 98
hp_coef() (*built-in function*), 248
hp_coef2() (*built-in function*), 249
hpr2() (*built-in function*), 267
hwfilter() (*built-in function*), 246
hwfilter2() (*built-in function*), 246

I

id_cmp() (*built-in function*), 87
ifirstloc() (*built-in function*), 65
imaxloc() (*built-in function*), 65
iminloc() (*built-in function*), 65
indent() (*built-in function*), 51
init_fft() (*built-in function*), 238
init_memt19937() (*built-in function*), 76
init_mt19937() (*built-in function*), 76
INTERFACES, 9, 13
inv() (*built-in function*), 195
is_alpha() (*built-in function*), 54
is_digit() (*built-in function*), 54
is_lower() (*built-in function*), 54
is_nan() (*built-in function*), 47
is_num() (*built-in function*), 54
is_same() (*built-in function*), 54
is_space() (*built-in function*), 54
is_upper() (*built-in function*), 54

L

lae2() (*built-in function*), 125
laev2() (*built-in function*), 120
lamch() (*built-in function*), 46
las2() (*built-in function*), 148
lascl() (*built-in function*), 71
lassq() (*built-in function*), 69
lassq2e() (*built-in function*), 70

lassqe() (*built-in function*), 70
LBLAS, 9, 10
LD_LIBRARY_PATH, 30
leapyr() (*built-in function*), 57
leave_proc() (*built-in function*), 51
leaving() (*built-in function*), 51
LIB, 9, 13, 14, 22
LIBTOOL, 9
LIBTOOLFLAGS, 9
lin_lu_solve() (*built-in function*), 194
LLAPACK, 9, 10
llsq_qr_solve() (*built-in function*), 185
llsq_qr_solve2() (*built-in function*), 185
llsq_svd_solve() (*built-in function*), 188
llsq_svd_solve2() (*built-in function*), 188
lngamma() (*built-in function*), 199
LOADFLAGS, 9–11, 30
lp_coef() (*built-in function*), 247
lp_coef2() (*built-in function*), 248
lq_cmp() (*built-in function*), 103
lu_cmp() (*built-in function*), 190
lu_cmp2() (*built-in function*), 190
lu_solve() (*built-in function*), 193
lu_solve2() (*built-in function*), 193

M

ma() (*built-in function*), 245
mach() (*built-in function*), 46
matmul2() (*built-in function*), 63
maxdiag_gkinv_ldu() (*built-in function*), 173
maxdiag_gkinv_qr() (*built-in function*), 172
maxdiag_tinv_ldu() (*built-in function*), 128
maxdiag_tinv_qr() (*built-in function*), 128
merror() (*built-in function*), 66
mid_shift() (*built-in function*), 56
MKL_NUM_THREADS, 33
mmproduct() (*built-in function*), 62
moddan_coef() (*built-in function*), 252
moddan_filter() (*built-in function*), 254
mvalloc() (*built-in function*), 65
my_date_time() (*built-in function*), 61

N

nan() (*built-in function*), 48
nbrchf() (*built-in function*), 56
NOOPTFLAGS, 9
norm() (*built-in function*), 69
norm2e() (*built-in function*), 70
normal_rand_number() (*built-in function*), 78
normal_rand_number2() (*built-in function*), 79
normal_rand_number3() (*built-in function*), 80
normal_random_number2_() (*built-in function*), 79
normal_random_number3_() (*built-in function*), 80
normal_random_number_() (*built-in function*), 79
norme() (*built-in function*), 69
nrm2() (*built-in function*), 266

O

obt_fmt() (*built-in function*), 56
OMP_DYNAMIC, 5, 32
OMP_MAX_ACTIVE_LEVELS, 5, 32
OMP_NESTED, 5, 32
OMP_NUM_THREADS, 5, 32
OMP_STACKSIZE, 5, 32
OPENBLAS_NUM_THREADS, 33
OPTFLAGS, 9
OPTS, 9, 10, 13, 16
orgbr() (*built-in function*), 271
orgtr() (*built-in function*), 269
ormbr() (*built-in function*), 271
ormtr() (*built-in function*), 269
ortho_gen_bd() (*built-in function*), 141
ortho_gen_bd2() (*built-in function*), 142
ortho_gen_lq() (*built-in function*), 103
ortho_gen_p_bd() (*built-in function*), 142
ortho_gen_q_bd() (*built-in function*), 142
ortho_gen_qr() (*built-in function*), 108
ortho_gen_qr_ts() (*built-in function*), 109
ortho_gen_random_qr() (*built-in function*), 81
ortho_gen_symtrid() (*built-in function*), 114
outerand() (*built-in function*), 68
outerdiff() (*built-in function*), 68
outerdiv() (*built-in function*), 68
outeror() (*built-in function*), 68
outerprod() (*built-in function*), 67
outersum() (*built-in function*), 68

P

partial_qr_cmp() (*built-in function*), 106
partial_qr_cmp_fixed_precision() (*built-in function*), 107
partial_rqr_cmp() (*built-in function*), 82
partial_rqr_cmp2() (*built-in function*), 83
partial_rqr_cmp_fixed_precision() (*built-in function*), 85
partial_rtqr_cmp() (*built-in function*), 85
permute_cor() (*built-in function*), 225
phase_scramble_cor() (*built-in function*), 225
pinvbeta() (*built-in function*), 202
pinvf2() (*built-in function*), 211
pinvgamma() (*built-in function*), 201
pinvn() (*built-in function*), 203
pinvn2() (*built-in function*), 204
pinvq() (*built-in function*), 209
pinvq2() (*built-in function*), 209

pinvstudent() (*built-in function*), 206
pinvt() (*built-in function*), 205
pk_coef() (*built-in function*), 252
poly() (*built-in function*), 67
poly_term() (*built-in function*), 67
posv() (*built-in function*), 272
power_spctrm() (*built-in function*), 259
power_spctrm2() (*built-in function*), 261
power_spectrum() (*built-in function*), 262
power_spectrum2() (*built-in function*), 264
print_array() (*built-in function*), 52
print_prinfac() (*built-in function*), 52
print_stat() (*built-in function*), 53
probbeta() (*built-in function*), 201
probbinom() (*built-in function*), 212
probf() (*built-in function*), 210
probf2() (*built-in function*), 211
probgamma() (*built-in function*), 199
probgamma2() (*built-in function*), 200
probgamma3() (*built-in function*), 200
probn() (*built-in function*), 202
probn2() (*built-in function*), 203
probq() (*built-in function*), 207
probq2() (*built-in function*), 207
probq3() (*built-in function*), 208
probstudent() (*built-in function*), 206
probt() (*built-in function*), 204
prodgiv() (*built-in function*), 126
prodgiv_eigvec() (*built-in function*), 127
product_svd_cmp() (*built-in function*), 181
put_diag() (*built-in function*), 71
pythag() (*built-in function*), 72
pythage() (*built-in function*), 72

Q

qlp_cmp() (*built-in function*), 163
qlp_cmp2() (*built-in function*), 164
qr_cmp() (*built-in function*), 104
qr_cmp2() (*built-in function*), 105
qr_cmp_ts() (*built-in function*), 104
qr_solve() (*built-in function*), 186
qr_solve2() (*built-in function*), 187
qrfac() (*built-in function*), 108
qrstep() (*built-in function*), 126
qrstep_bd() (*built-in function*), 178
qrstep_zero_bd() (*built-in function*), 179
quick_sort() (*built-in function*), 49

R

rand_integer31() (*built-in function*), 78
rand_integer32() (*built-in function*), 77
rand_number() (*built-in function*), 77
random_integer31_() (*built-in function*), 78
random_integer32_() (*built-in function*), 78
random_number_() (*built-in function*), 77
random_qr_cmp() (*built-in function*), 80
random_seed_() (*built-in function*), 76
rangen() (*built-in function*), 212
rank() (*built-in function*), 49
real_fft() (*built-in function*), 240
real_fft_backward() (*built-in function*), 240
real_fft_forward() (*built-in function*), 240
realloc() (*built-in function*), 73
reallocate() (*built-in function*), 72
reig_cmp() (*built-in function*), 120
reig_pos_cmp() (*built-in function*), 162
reorder() (*built-in function*), 49
replace_nan() (*built-in function*), 47
rot() (*built-in function*), 266
rot_givens() (*built-in function*), 91
rqb_cmp() (*built-in function*), 86
rqb_cmp_fixed_precision() (*built-in function*), 87
rqb_solve() (*built-in function*), 187
rqlp_cmp() (*built-in function*), 164
rqlp_svd_cmp() (*built-in function*), 158
rqlp_svd_cmp2() (*built-in function*), 159
rqlp_svd_cmp_fixed_precision() (*built-in function*), 161
rqr_svd_cmp() (*built-in function*), 157
rqr_svd_cmp_fixed_precision() (*built-in function*), 159
rsvd_cmp() (*built-in function*), 157
rsvd_cmp_fixed_precision() (*built-in function*), 160
rtsw() (*built-in function*), 59

S

scal() (*built-in function*), 266
scatter_add() (*built-in function*), 70
scatter_max() (*built-in function*), 70
select_eigval_cmp() (*built-in function*), 123
select_eigval_cmp2() (*built-in function*), 124
select_eigval_cmp3() (*built-in function*), 124
select_singval_cmp() (*built-in function*), 166
select_singval_cmp2() (*built-in function*), 167
select_singval_cmp3() (*built-in function*), 169
select_singval_cmp4() (*built-in function*), 170
simple_shuffle() (*built-in function*), 89
singval_sort() (*built-in function*), 171
singvalues() (*built-in function*), 165
singvec_sort() (*built-in function*), 172
solve_lin() (*built-in function*), 194
solve_llsq() (*built-in function*), 184
spctrm_diff() (*built-in function*), 258
spctrm_diff2() (*built-in function*), 259
spctrm_ratio() (*built-in function*), 256
spctrm_ratio2() (*built-in function*), 256

spctrm_ratio3() (*built-in function*), 256
spctrm_ratio4() (*built-in function*), 257
spev() (*built-in function*), 269
spevd() (*built-in function*), 269
spevx() (*built-in function*), 270
spmv() (*built-in function*), 266
spr2() (*built-in function*), 267
STATPACKDIR, 8
STATPACKDIR, 8
stedc() (*built-in function*), 270
stemr() (*built-in function*), 270
steqr() (*built-in function*), 270
stev() (*built-in function*), 270
stevd() (*built-in function*), 270
stevr() (*built-in function*), 270
stevx() (*built-in function*), 270
string_comp() (*built-in function*), 55
string_count() (*built-in function*), 55
string_eq() (*built-in function*), 55
string_index() (*built-in function*), 55
string_to_val() (*built-in function*), 56
svd_cmp() (*built-in function*), 148
svd_cmp2() (*built-in function*), 149
svd_cmp3() (*built-in function*), 150
svd_cmp4() (*built-in function*), 151
svd_cmp5() (*built-in function*), 153
svd_cmp6() (*built-in function*), 154
svd_cmp7() (*built-in function*), 155
svd_cmp8() (*built-in function*), 156
svd_sort() (*built-in function*), 172
svd_sort2() (*built-in function*), 172
swap() (*built-in function*), 64, 266
syev() (*built-in function*), 269
syevd() (*built-in function*), 269
syevr() (*built-in function*), 269
syevx() (*built-in function*), 269
sygv() (*built-in function*), 270
sygvd() (*built-in function*), 270
sygvx() (*built-in function*), 270
sym_inv() (*built-in function*), 196
sym_trid_cmp() (*built-in function*), 197
sym_trid_cmp2() (*built-in function*), 198
sym_trid_solve() (*built-in function*), 198
symlin_filter() (*built-in function*), 253
symlin_filter2() (*built-in function*), 253
symm() (*built-in function*), 267
symtrid_bisect() (*built-in function*), 117
symtrid_cmp() (*built-in function*), 113
symtrid_cmp2() (*built-in function*), 114
symtrid_deflate() (*built-in function*), 127
symtrid_qri() (*built-in function*), 115
symtrid_qri2() (*built-in function*), 115
symtrid_qri3() (*built-in function*), 116
symtrid_ratqri() (*built-in function*), 116
symtrid_ratqri2() (*built-in function*), 117
symv() (*built-in function*), 266
syr2() (*built-in function*), 267
syr2k() (*built-in function*), 267
syrk() (*built-in function*), 267
system_date_time() (*built-in function*), 61
sysv() (*built-in function*), 272
sytrd() (*built-in function*), 269

T

taper() (*built-in function*), 254
test_ieee() (*built-in function*), 46
test_nan() (*built-in function*), 47
time_to_hmsms() (*built-in function*), 60
time_to_string() (*built-in function*), 60
to_lower() (*built-in function*), 56
to_upper() (*built-in function*), 55
tpmqrt() (*built-in function*), 273
tpqrt() (*built-in function*), 273
transpose2() (*built-in function*), 62
tri_insert() (*built-in function*), 48
triang_solve() (*built-in function*), 195
triangle() (*built-in function*), 68
trid_cmp() (*built-in function*), 129
trid_cmp2() (*built-in function*), 130
trid_deflate() (*built-in function*), 127
trid_inviter() (*built-in function*), 131
trid_qr_cmp() (*built-in function*), 129
trid_qr_solve() (*built-in function*), 129
trid_solve() (*built-in function*), 130
trmm() (*built-in function*), 267
trsm() (*built-in function*), 267
trsv() (*built-in function*), 266
true_nan() (*built-in function*), 48
ts_id_cmp() (*built-in function*), 88

U

unit_matrix() (*built-in function*), 71
update_cor() (*built-in function*), 226
update_cor_miss2() (*built-in function*), 226
update_mvs() (*built-in function*), 217
update_mvs_grp() (*built-in function*), 219
update_mvs_grp_miss() (*built-in function*), 220
update_rk1() (*built-in function*), 67
update_rk2() (*built-in function*), 67
upper_bd_deflate() (*built-in function*), 179
upper_bd_dpqd() (*built-in function*), 176
upper_bd_dpqd2() (*built-in function*), 177
upper_bd_dsqd() (*built-in function*), 176
upper_bd_dsqd2() (*built-in function*), 177

V

val_to_string() (*built-in function*), 56
valmed() (*built-in function*), 222

W

write_array() (*built-in function*), 52

Y

ymd_to_daynum() (*built-in function*), 58
ymd_to_dayweek() (*built-in function*), 58

Z

zroots_unity() (*built-in function*), 67