intel®

Intel® Math Kernel Library

Reference Manual

Document Number: 630813-017

World Wide Web: http://developer.intel.com

Version	Version Information	Date
-001	Original Issue.	11/94
-002	Added functions crotg, zrotg. Documented versions of functions ?her2k, ?symm, ?syrk, and ?syr2k not previously described. Pagination revised.	5/95
-003	Changed the title; former title: “Intel BLAS Library for the Pentium® Processor Reference Manual.” Added functions ?rotm,?rotmg and updated Appendix C.	1/96
-004	Documents Intel® Math Kernel library (Intel® MKL) release 2.0 with the parallelism capability. Information on parallelism has been added in Chapter 1 and in section “BLAS Level 3 Routines” in Chapter 2.	11/96
-005	Two-dimensional FFTs have been added. C interface has been added to both one- and two-dimensional FFTs.	8/97
-006	Documents Intel Math Kernel Library release 2.1. Sparse BLAS section has been added in Chapter 2.	1/98
-007	Documents Intel Math Kernel Library release 3.0. Descriptions of LAPACK routines (Chapters 4 and 5) and CBLAS interface (Appendix C) have been added. Quick Reference has been excluded from the manual; MKL 3.0 Quick Reference is now available in HTML format.	1/99
-008	Documents Intel Math Kernel Library release 3.2. Description of FFT routines have been revised. In Chapters 4 and 5 NAG names for LAPACK routines have been excluded.	6/99
-009	New LAPACK routines for eigenvalue problems have been added inchapter 5.	11/99
-010	Documents Intel Math Kernel Library release 4.0. Chapter 6 describing the VML functions has been added.	06/00
-011	Documents Intel Math Kernel Library release 5.1. LAPACK section has been extended to include the full list of computational and driver routines .	04/01
-6001	Documents Intel Math Kernel Library release 6.0 beta. New DFT interface and Vector Statistical Library functions have been added.	07/02
-6002	Documents Intel Math Kernel Library 6.0 beta update. DFT functions description has been updated. CBLAS interface description was extended.	12/02
-6003	Documents Intel Math Kernel Library 6.0 gold. DFT functions have been updated. Auxiliary LAPACK routines’ descriptions were added to the manual.	03/03
-6004	Documents Intel Math Kernel Library release 6.1.	07/03
-6005	Documents Intel Math Kernel Library release 7.0 beta. Includes ScaLAPACK and sparse solver descriptions.	11/03
-017	Documents Intel MKL and Intel® Cluster MKL release 7.0 gold. Auxiliary ScaLAPACK and alternative sparse solver interface were added.	04/04

Contents

Chapter 3 LAPACK Routines: Linear Equations

Chapter 4 LAPACK Routines: Least Squares and Eigenvalue Problems

Chapter 5 LAPACK Auxiliary and Utility Routines

Overview 1

The Intel® Math Kernel Library (Intel® MKL) provides Fortran routines and functions that perform a wide variety of operations on vectors and matrices including sparse matrices. The library also includes discrete Fourier transform routines, as well as vector mathematical and vector statistical functions with Fortran and C interfaces.

The version of the library named Intel® Cluster MKL is a superset of Intel MKL and includes also ScaLAPACK software for solving linear algebra problems on distributed-memory parallel computers.

The Intel MKL enhances performance of the application programs that use it because the library has been optimized for latest generations of Intel® processors.
This chapter introduces the Intel Math Kernel Library and provides information about the organization of this manual.

About This Software

The Intel Math Kernel Library includes the following groups of routines:

- Basic Linear Algebra Subprograms (BLAS):
  – vector operations
  – matrix-vector operations
  – matrix-matrix operations
- Sparse BLAS (basic vector operations on sparse vectors)
- LAPACK routines for solving systems of linear equations
- LAPACK routines for solving least-squares problems, eigenvalue and singular value problems, and Sylvester's equations
- Auxiliary and utility LAPACK routines

- ScaLAPACK computational, driver and auxiliary routines (for Intel Cluster MKL only)
- Direct Sparse Solver routines
- Vector Mathematical Library (VML) functions for computing core mathematical functions on vector arguments (with Fortran and C interfaces)
- Vector Statistical Library (VSL) functions for generating vectors of pseudorandom numbers with different types of statistical distributions
- General Discrete Fourier Transform Functions (DFT) and a subset of Fast Fourier transform routines (FFT) with Fortran and C interfaces.

For specific issues on using the library, please refer to the *MKL Release Notes*.

Technical Support

Intel MKL provides a product web site that offers timely and comprehensive product information, including product features, white papers, and technical articles. For the latest information, check: http://developer.intel.com/software/products/

Intel also provides a support web site that contains a rich repository of self help information, including getting started tips, known product issues, product errata, license information, user forums, and more (visit http://support.intel.com/support/).

Registering your product entitles you to one year of technical support and product updates through Intel® Premier Support. Intel Premier Support is an interactive issue management and communication web site providing these services:

- Submit issues and review their status.
- Download product updates anytime of the day.

To register your product, contact Intel, or seek product support, please visit: http://www.intel.com/software/products/support

BLAS Routines

BLAS routines and functions are divided into the following groups according to the operations they perform:

- BLAS Level 1 Routines and Functions perform operations of both addition and reduction on vectors of data. Typical operations include scaling and dot products.
- BLAS Level 2 Routines perform matrix-vector operations, such as matrix-vector multiplication, rank-1 and rank-2 matrix updates, and solution of triangular systems.

- BLAS Level 3 Routines perform matrix-matrix operations, such as matrix-matrix multiplication, rank-k update, and solution of triangular systems.

Sparse BLAS Routines

Sparse BLAS Routines and Functions operate on sparse vectors (that is, vectors in which most of the elements are zeros). These routines perform vector operations similar to BLAS Level 1 routines. Sparse BLAS routines take advantage of vectors' sparsity: they allow you to store only non-zero elements of vectors.

LAPACK Routines

The Intel Math Kernel Library covers the full set of the LAPACK computational, driver, auxiliary and utility routines.

The original versions of LAPACK from which that part of Intel MKL was derived can be obtained from http://www.netlib.org/lapack/index.html. The authors of LAPACK are E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen.

The LAPACK routines can be divided into the following groups according to the operations they perform:

- Routines for solving systems of linear equations, factoring and inverting matrices, and estimating condition numbers (see Chapter 3).
- Routines for solving least-squares problems, eigenvalue and singular value problems, and Sylvester's equations (see Chapter 4).
- Auxiliary and utility routines used to perform certain subtasks, common low-level computation or related tasks (see Chapter 5).

ScaLAPACK Routines

ScaLAPACK package (included with Intel Cluster MKL only, see Chapter 6 and Chapter 7) runs on distributed-memory architectures and includes routines for solving systems of linear equations, solving linear least-squares problems, eigenvalue and singular value problems, as well as performing a number of related computational tasks.

The original versions of ScaLAPACK from which that part of Intel Cluster MKL was derived can be obtained from http://www.netlib.org/scalapack/index.html. The authors of ScaLAPACK are L. Blackford, J. Choi, A.Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K.Stanley, D. Walker, and R. Whaley.

Intel Cluster MKL version of ScaLAPACK is optimized for Intel processors and uses MPICH version of MPI.

Sparse Solver Routines

Direct sparse solver routines in Intel MKL (see Chapter 8) solve symmetric and symmetrically-structured sparse matrices with real or complex coefficients. For symmetric matrices, these Intel MKL subroutines can solve both positive definite and indefinite systems. Intel MKL includes the PARDISO* sparse solver interface as well as an alternative set of user callable direct sparse solver routines.

VML Functions

Vector Mathematical Library (VML) functions (see Chapter 9) include a set of highly optimized implementations of certain computationally expensive core mathematical functions (power, trigonometric, exponential, hyperbolic etc.) that operate on real vector arguments.

VSL Functions

Vector Statistical Library (VSL) functions (see Chapter 10) include a set of pseudo- and quasi-random number generator subroutines implementing basic continuous and discrete distributions. To provide best performance, VSL subroutines use calls to highly optimized Basic Random Number Generators and the library of vector mathematical functions, VML.

DFT and FFT Functions

The Intel MKL multidimensional Discrete Fourier Transform functions with mixed radix support (see Chapter 11) provide uniformity of DFT computation and combine functionality with ease of use. Both Fortran and C interface specification are given.

For compatibility with previous versions, Intel MKL provides also a set of simplified one- and two-dimensional Fast Fourier Transform functions (see Chapter 12) that support powers of 2 transform size.

Performance Enhancements

The Intel Math Kernel Library has been optimized by exploiting both processor and system features and capabilities. Special care has been given to those routines that most profit from cache-management techniques. These especially include matrix-matrix operation routines such as `dgemm()`.

In addition, code optimization techniques have been applied to minimize dependencies of scheduling integer and floating-point units on the results within the processor.

The major optimization techniques used throughout the library include:

- Loop unrolling to minimize loop management costs.
- Blocking of data to improve data reuse opportunities.
- Copying to reduce chances of data eviction from cache.
- Data prefetching to help hide memory latency.
- Multiple simultaneous operations (for example, dot products in `dgemm`) to eliminate stalls due to arithmetic unit pipelines.
- Use of hardware features such as the SIMD arithmetic units, where appropriate.

These are techniques from which the arithmetic code benefits the most.

Parallelism

In addition to the performance enhancements discussed above, the Intel MKL offers performance gains through parallelism provided by the symmetric multiprocessing performance (SMP) feature. You can obtain improvements from SMP in the following ways:

- One way is based on user-managed threads in the program and further distribution of the operations over the threads based on data decomposition, domain decomposition, control decomposition, or some other parallelizing technique. Each thread can use any of the Intel MKL functions because the library has been designed to be thread-safe.

- Another method is to use the FFT and BLAS level 3 routines. They have been parallelized and require no alterations of your application to gain the performance enhancements of multiprocessing. Performance using multiple processors on the level 3 BLAS shows excellent scaling. Since the threads are called and managed within the library, the application does not need to be recompiled thread-safe (see also BLAS Level 3 Routines in Chapter 2).
- Yet another method is to use *tuned LAPACK routines*. Currently these include the single- and double precision flavors of routines for *QR* factorization of general matrices, triangular factorization of general and symmetric positive-definite matrices, solving systems of equations with such matrices, as well as solving symmetric eigenvalue problems.

For instructions on setting the number of available processors for the BLAS level 3 and LAPACK routines, see the *Intel MKL Technical User Notes*.

Platforms Supported

The Intel Math Kernel Library includes Fortran routines and functions optimized for Intel® processor-based computers running operating systems that support multiprocessing. In addition to the Fortran interface, the Intel MKL includes a C-language interface for the Discrete Fourier transform functions, as well as for the Vector Mathematical Library and Vector Statistical Library functions.
For hardware and software requirements to use Inlel MKL, see *MKL Release Notes.*

About This Manual

This manual describes the routines and functions of the Intel MKL and Intel Cluster MKL. Each reference section describes a routine group typically consisting of routines used with four basic data types: single-precision real, double-precision real, single-precision complex, and double-precision complex.

Each routine group is introduced by its name, a short description of its purpose, and the calling sequence, or syntax, for each type of data with which each routine of the group is used. The following sections are also included:

Description — Describes the operation performed by routines of the group based on one or more equations. The data types of the arguments are defined in general terms for the group.

Input Parameters — Defines the data type for each parameter on entry, for example:

`a` `REAL` for `saxpy`
`DOUBLE PRECISION` for `daxpy`

Output Parameters	Lists resultant parameters on exit.

Audience for This Manual

The manual addresses programmers proficient in computational mathematics and assumes a working knowledge of the principles and vocabulary of linear algebra, mathematical statistics, and Fourier transforms.

Manual Organization

The manual contains the following chapters and appendixes:

Chapter 1	Overview. Introduces the Intel Math Kernel Library software, provides information on manual organization, and explains notational conventions.
Chapter 2	BLAS and Sparse BLAS Routines. Provides descriptions of BLAS and Sparse BLAS functions and routines.
Chapter 3	LAPACK Routines: Linear Equations. Provides descriptions of LAPACK routines for solving systems of linear equations and performing a number of related computational tasks: triangular factorization, matrix inversion, estimating the condition number of matrices.
Chapter 4	LAPACK Routines: Least Squares and Eigenvalue Problems. Provides descriptions of LAPACK routines for solving least-squares problems, standard and generalized eigenvalue problems, singular value problems, and Sylvester's equations.
Chapter 5	LAPACK Auxiliary and Utility Routines. Describes auxiliary and utility LAPACK routines that perform certain subtasks or common low-level computation.
Chapter 6	ScaLAPACK Routines. Describes ScaLAPACK computational and driver routines (software included with Intel Cluster MKL only).
Chapter 7	ScaLAPACK Auxiliary and Utility Routines. Describes ScaLAPACK auxiliary routines (software included with Intel Cluster MKL only).
Chapter 8	Sparse Solver Routines. Describes direct sparse solver routines that solve symmetric and symmetrically-structured sparse matrices.
Chapter 9	Vector Mathematical Functions. Provides descriptions of VML functions for computing elementary mathematical functions on vector arguments.

Chapter 10	Vector Generators of Statistical Distributions. Provides descriptions of VSL functions for generating vectors of pseudorandom numbers.
Chapter 11	Discrete Fourier Transform Functions. Describes multidimensional functions for computing the Discrete Fourier Transform.
Chapter 12	Fast Fourier Transforms. Provides descriptions of a simplified fast Fourier transform (FFT) routines.
Appendix A	Linear Solvers Basics. Briefly describes the basic definitions and approaches used in linear algebra for solving systems of linear equations.
Appendix B	Routine and Function Arguments. Describes the major arguments of the BLAS routines and VML functions: vector and matrix arguments.
Appendix C	Code Examples. Provides code examples of calling various Intel MKL functions and routines (BLAS, Sparse Solver, DFT).
Appendix D	CBLAS Interface to the BLAS. Provides the C interface to the BLAS.

The manual also includes a Bibliography, Glossary and an Index.

Notational Conventions

This manual uses the following notational conventions:

- Routine name shorthand (`?ungqr` instead of `cungqr/zungqr`).
- Font conventions used for distinction between the text and the code.

Routine Name Shorthand

For shorthand, character codes are represented by a question mark "`?`" in names of routine groups. The question mark is used to indicate any or all possible varieties of a function; for example:

`?swap` Refers to all four data types of the vector-vector `?swap` routine: `sswap`, `dswap`, `cswap`, and `zswap`.

Font Conventions

The following font conventions are used:

`UPPERCASE COURIER` Data type used in the discussion of input and output parameters for Fortran interface. For example, `CHARACTER*1`.

`lowercase courier`	Code examples: `a(k+i,j) = matrix(i,j)` and data types for C interface, for example, `const float*`
`lowercase courier mixed with UpperCase courier`	Function names for C interface, for example, `vmlSetMode`
`lowercase courier italic`	Variables in arguments and parameters discussion. For example, *`incx`*.
`*`	Used as a multiplication symbol in code examples and equations and where required by the Fortran syntax.

BLAS and Sparse BLAS Routines 2

This chapter contains descriptions of the BLAS and Sparse BLAS routines of the Intel® Math Kernel Library. The routine descriptions are arranged in four sections according to the BLAS level of operation:

- "BLAS Level 1 Routines and Functions" (vector-vector operations)
- BLAS Level 2 Routines (matrix-vector operations)
- BLAS Level 3 Routines (matrix-matrix operations)
- Sparse BLAS Routines and Functions.

Each section presents the routine and function group descriptions in alphabetical order by routine or function group name; for example, the `?asum` group, the `?axpy` group. The question mark in the group name corresponds to different character codes indicating the data type (`s`, `d`, `c`, and `z` or their combination); see *Routine Naming Conventions* on the next page.

When BLAS routines encounter an error, they call the error reporting routine `xerbla`. To be able to view error reports, you must include `xerbla` in your code. A copy of the source code for `xerbla` is included in the library.

In BLAS Level 1 groups `i?amax` and `i?amin`, an "i" is placed before the character code and corresponds to the index of an element in the vector. These groups are placed in the end of the BLAS Level 1 section.

Routine Naming Conventions

BLAS routine names have the following structure:

`<character code> <name> <mod> ( )`

The `<character code>` is a character that indicates the data type:

`s`	real, single precision	`c`	complex, single precision
`d`	real, double precision	`z`	complex, double precision

Some routines and functions can have combined character codes, such as `sc` or `dz`. For example, the function `scasum` uses a complex input array and returns a real value.

The `<name>` field, in BLAS level 1, indicates the operation type. For example, the BLAS level 1 routines `?dot`, `?rot`, `?swap` compute a vector dot product, vector rotation, and vector swap, respectively.

In BLAS level 2 and 3, `<name>` reflects the matrix argument type:

`ge` general matrix
`gb` general band matrix
`sy` symmetric matrix
`sp` symmetric matrix (packed storage)
`sb` symmetric band matrix
`he` Hermitian matrix
`hp` Hermitian matrix (packed storage)
`hb` Hermitian band matrix
`tr` triangular matrix
`tp` triangular matrix (packed storage)
`tb` triangular band matrix.

The `<mod>` field, if present, provides additional details of the operation. BLAS level 1 names can have the following characters in the `<mod>` field:

`c` conjugated vector
`u` unconjugated vector
`g` Givens rotation.

BLAS level 2 names can have the following characters in the `<mod>` field:
`mv` matrix-vector product
`sv` solving a system of linear equations with matrix-vector operations
`r` rank-1 update of a matrix
`r2` rank-2 update of a matrix.

BLAS level 3 names can have the following characters in the <*mod*> field:

`mm` matrix-matrix product
`sm` solving a system of linear equations with matrix-matrix operations
`rk` rank-*k* update of a matrix
`r2k` rank-2*k* update of a matrix.

The examples below illustrate how to interpret BLAS routine names:

`<d> <dot>`	`ddot`: double-precision real vector-vector dot product
`<c> <dot> <c>`	`cdotc`: complex vector-vector dot product, conjugated
`<sc> <asum>`	`scasum`: sum of magnitudes of vector elements, single precision real output and single precision complex input
`<c> <dot> <u>`	`cdotu`: vector-vector dot product, unconjugated, complex
`<s> <ge> <mv>`	`sgemv`: matrix-vector product, general matrix, single precision
`<z> <tr> <mm>`	`ztrmm`: matrix-matrix product, triangular matrix, double-precision complex.

Sparse BLAS naming conventions are similar to those of BLAS level 1.
For more information, see "Naming Conventions in Sparse BLAS".

Matrix Storage Schemes

Matrix arguments of BLAS routines can use the following storage schemes:

- *Full storage*: a matrix A is stored in a two-dimensional array `a`, with the matrix element a_{ij} stored in the array element `a(i,j)`.
- *Packed storage* scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly: the upper or lower triangle of the matrix is packed by columns in a one-dimensional array.
- *Band storage*: a band matrix is stored compactly in a two-dimensional array: columns of the matrix are stored in the corresponding columns of the array, and *diagonals* of the matrix are stored in rows of the array.

For more information on matrix storage schemes, see "Matrix Arguments" in Appendix B.

BLAS Level 1 Routines and Functions

BLAS Level 1 includes routines and functions, which perform vector-vector operations. Table 2-1 lists the BLAS Level 1 routine and function groups and the data types associated with them.

Table 2-1 BLAS Level 1 Routine Groups and Their Data Types

Routine or Function Group	Data Types	Description
?asum	s, d, sc, dz	Sum of vector magnitudes (functions)
?axpy	s, d, c, z	Scalar-vector product (routines)
?copy	s, d, c, z	Copy vector (routines)
?dot	s, d	Dot product (functions)
?sdot	sd, d	Dot product with extended precision (functions)
?dotc	c, z	Dot product conjugated (functions)
?dotu	c, z	Dot product unconjugated (functions)
?nrm2	s, d, sc, dz	Vector 2-norm (Euclidean norm) a normal or null vector (functions)
?rot	s, d, cs, zd	Plane rotation of points (routines)
?rotg	s, d, c, z	Givens rotation of points (routines)
?rotm	s, d	Modified plane rotation of points
?rotmg	s, d	Givens modified plane rotation of points
?scal	s, d, c, z, cs, zd	Vector scaling (routines)
?swap	s, d, c, z	Vector-vector swap (routines)
i?amax	s, d, c, z	Vector maximum value, absolute largest element of a vector where *i* is an index to this value in the vector array (functions)
i?amin	s, d, c, z	Vector minimum value, absolute smallest element of a vector where *i* is an index to this value in the vector array (functions)

?asum

Computes the sum of magnitudes of the vector elements.

Syntax

```
res = sasum ( n, x, incx )
res = scasum ( n, x, incx )
res = dasum ( n, x, incx )
res = dzasum ( n, x, incx )
```

Description

Given a vector *x*, `?asum` functions compute the sum of the magnitudes of its elements or, for complex vectors, the sum of magnitudes of the elements' real parts plus magnitudes of their imaginary parts:

$$res = |\mathrm{Re}\,x(1)| + |\mathrm{Im}\,x(1)| + |\mathrm{Re}\,x(2)| + |\mathrm{Im}\,x(2)| + \ldots + |\mathrm{Re}\,x(n)| + |\mathrm{Im}\,x(n)|$$

where *x* is a vector of order *n*.

Input Parameters

n	`INTEGER`. Specifies the order of vector *x*.
x	`REAL` for `sasum` `DOUBLE PRECISION` for `dasum` `COMPLEX` for `scasum` `DOUBLE COMPLEX` for `dzasum` Array, `DIMENSION` at least `(1 + (n-1)*abs(incx))`.
incx	`INTEGER`. Specifies the increment for the elements of *x*.

Output Parameters

res	`REAL` for `sasum` `DOUBLE PRECISION` for `dasum` `REAL` for `scasum` `DOUBLE PRECISION` for `dzasum` Contains the sum of magnitudes of all elements' real parts plus magnitudes of their imaginary parts.

?axpy

Computes a vector-scalar product and adds the result to a vector.

Syntax

```
call saxpy ( n, a, x, incx, y, incy )
call daxpy ( n, a, x, incx, y, incy )
call caxpy ( n, a, x, incx, y, incy )
call zaxpy ( n, a, x, incx, y, incy )
```

Description

The ?axpy routines perform a vector-vector operation defined as

*y := a*x + y*

where:

a is a scalar

x and *y* are vectors of order *n*.

Input Parameters

n	INTEGER. Specifies the order of vectors *x* and *y*.
a	REAL for saxpy DOUBLE PRECISION for daxpy COMPLEX for caxpy DOUBLE COMPLEX for zaxpy Specifies the scalar *a*.
x	REAL for saxpy DOUBLE PRECISION for daxpy COMPLEX for caxpy DOUBLE COMPLEX for zaxpy Array, DIMENSION at least (1 + (*n*-1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for the elements of *x*.

y	`REAL` for `saxpy` `DOUBLE PRECISION` for `daxpy` `COMPLEX` for `caxpy` `DOUBLE COMPLEX` for `zaxpy` Array, `DIMENSION` at least `(1 + (n-1)*abs(incy))`.
incy	`INTEGER`. Specifies the increment for the elements of *y*.

Output Parameters

y	Contains the updated vector *y*.

?copy

Copies vector to another vector.

Syntax

```
call scopy ( n, x, incx, y, incy )
call dcopy ( n, x, incx, y, incy )
call ccopy ( n, x, incx, y, incy )
call zcopy ( n, x, incx, y, incy )
```

Description

The `?copy` routines perform a vector-vector operation defined as

$y = x$

where *x* and *y* are vectors.

Input Parameters

n	`INTEGER`. Specifies the order of vectors *x* and *y*.
x	`REAL` for `scopy` `DOUBLE PRECISION` for `dcopy` `COMPLEX` for `ccopy` `DOUBLE COMPLEX` for `zcopy` Array, `DIMENSION` at least `(1 + (n-1)*abs(incx))`.

incx	`INTEGER`. Specifies the increment for the elements of *x*.
y	`REAL` for `scopy` `DOUBLE PRECISION` for `dcopy` `COMPLEX` for `ccopy` `DOUBLE COMPLEX` for `zcopy` Array, `DIMENSION` at least `(1 + (n-1)*abs(incy))`.
incy	`INTEGER`. Specifies the increment for the elements of *y*.

Output Parameters

y	Contains a copy of the vector *x* if *n* is positive. Otherwise, parameters are unaltered.

?dot

Computes a vector-vector dot product.

Syntax

```
res = sdot ( n, x, incx, y, incy )
res = ddot ( n, x, incx, y, incy )
```

Description

The `?dot` functions perform a vector-vector reduction operation defined as

$$res = \sum (x*y),$$

where *x* and *y* are vectors.

Input Parameters

n	`INTEGER`. Specifies the order of vectors *x* and *y*.
x	`REAL` for `sdot` `DOUBLE PRECISION` for `ddot` Array, `DIMENSION` at least `(1+(n-1)*abs(incx))`.
incx	`INTEGER`. Specifies the increment for the elements of *x*.

y REAL for sdot
DOUBLE PRECISION for ddot

Array, DIMENSION at least (1+(*n*-1)*abs(*incy*)).

incy INTEGER. Specifies the increment for the elements of *y*.

Output Parameters

res REAL for sdot
DOUBLE PRECISION for ddot

Contains the result of the dot product of *x* and *y*, if *n* is positive. Otherwise, *res* contains 0.

?sdot

Computes a vector-vector dot product with extended precision.

Syntax

```
res = sdsdot ( n, sb, sx, incx, sy, incy )
res = dsdot ( n, sx, incx, sy, incy )
```

Description

The ?sdot functions compute the inner product of two vectors with extended precision. Both functions use extended precision accumulation of the intermediate results, but the function sdsdot outputs the final result in single precision, whereas the function dsdot outputs the double precision result. The function sdsdot also adds scalar value *sb* to the inner product.

Input Parameters

n INTEGER. Specifies the number of elements in the input vectors *sx* and *sy*.

sb REAL. Single precision scalar to be added to inner product (for the function sdsdot only).

sx, sy REAL. Arrays, DIMENSION at least (1+(*n*-1)*abs(*incx*)) and (1+(*n*-1)*abs(*incy*)), respectively. Contain the input single precision vectors.

incx — INTEGER. Specifies the increment for the elements of *sx*.

incy — INTEGER. Specifies the increment for the elements of *sy*.

Output Parameters

res — REAL for sdsdot
DOUBLE PRECISION for dsdot

Contains the result of the dot product of *sx* and *sy* (with *sb* added for sdsdot), if *n* is positive. Otherwise, *res* contains *sb* for sdsdot and 0 for dsdot.

?dotc

Computes a dot product of a conjugated vector with another vector.

Syntax

```
res = cdotc ( n, x, incx, y, incy )
res = zdotc ( n, x, incx, y, incy )
```

Description

The ?dotc functions perform a vector-vector operation defined as

$$res = \sum (conjg(x)*y),$$

where *x* and *y* are *n*-element vectors.

Input Parameters

n — INTEGER. Specifies the order of vectors *x* and *y*.

x — COMPLEX for cdotc
DOUBLE COMPLEX for zdotc

Array, DIMENSION at least (1 + (*n*-1)*abs(*incx*)).

incx — INTEGER. Specifies the increment for the elements of *x*.

y `COMPLEX` for `cdotc`
`DOUBLE COMPLEX` for `zdotc`

Array, `DIMENSION` at least `(1 + (n-1)*abs(incy))`.

incy `INTEGER`. Specifies the increment for the elements of *y*.

Output Parameters

res `COMPLEX` for `cdotc`
`DOUBLE COMPLEX` for `zdotc`

Contains the result of the dot product of the conjugated *x* and unconjugated *y*, if *n* is positive. Otherwise, *res* contains `0`.

?dotu

Computes a vector-vector dot product.

Syntax

```
res = cdotu ( n, x, incx, y, incy )
res = zdotu ( n, x, incx, y, incy )
```

Description

The `?dotu` functions perform a vector-vector reduction operation defined as $res = \sum(x*y)$, where *x* and *y* are *n*-element complex vectors.

Input Parameters

n `INTEGER`. Specifies the order of vectors *x* and *y*.

x `COMPLEX` for `cdotu`
`DOUBLE COMPLEX` for `zdotu`

Array, `DIMENSION` at least `(1 + (n-1)*abs(incx))`.

incx `INTEGER`. Specifies the increment for the elements of *x*.

y `COMPLEX` for `cdotu`
`DOUBLE COMPLEX` for `zdotu`

Array, `DIMENSION` at least `(1 + (n-1)*abs(incy))`.

incy — `INTEGER`. Specifies the increment for the elements of *y*.

Output Parameters

res — `COMPLEX` for `cdotu`
`DOUBLE COMPLEX` for `zdotu`

Contains the result of the dot product of *x* and *y*, if *n* is positive. Otherwise, *res* contains `0`.

?nrm2

Computes the Euclidean norm of a vector.

Syntax

```
res = snrm2 ( n, x, incx )
res = dnrm2 ( n, x, incx )
res = scnrm2 ( n, x, incx )
res = dznrm2 ( n, x, incx )
```

Description

The `?nrm2` functions perform a vector reduction operation defined as

res = ||*x*||,

where:

x is a vector

res is a value containing the Euclidean norm of the elements of *x*.

Input Parameters

n	`INTEGER`. Specifies the order of vector `x`.
x	`REAL` for `snrm2` `DOUBLE PRECISION` for `dnrm2` `COMPLEX` for `scnrm2` `DOUBLE COMPLEX` for `dznrm2` Array, `DIMENSION` at least `(1 + (`*n*`-1)*abs(`*incx*`))`.
incx	`INTEGER`. Specifies the increment for the elements of *x*.

Output Parameters

res	`REAL` for `snrm2` `DOUBLE PRECISION` for `dnrm2` `REAL` for `scnrm2` `DOUBLE PRECISION` for `dznrm2` Contains the Euclidean norm of the vector *x*.

?rot

Performs rotation of points in the plane.

Syntax

```
call srot ( n, x, incx, y, incy, c, s )
call drot ( n, x, incx, y, incy, c, s )
call csrot ( n, x, incx, y, incy, c, s )
call zdrot ( n, x, incx, y, incy, c, s )
```

Description

Given two complex vectors *x* and *y*, each vector element of these vectors is replaced as follows:

```
x(i) = c*x(i) + s*y(i)
y(i) = c*y(i) - s*x(i)
```

Input Parameters

n	`INTEGER`. Specifies the order of vectors *x* and *y*.
x	`REAL` for `srot` `DOUBLE PRECISION` for `drot` `COMPLEX` for `csrot` `DOUBLE COMPLEX` for `zdrot` Array, `DIMENSION` at least `(1 + (n-1)*abs(incx))`.
incx	`INTEGER`. Specifies the increment for the elements of *x*.
y	`REAL` for `srot` `DOUBLE PRECISION` for `drot` `COMPLEX` for `csrot` `DOUBLE COMPLEX` for `zdrot` Array, `DIMENSION` at least `(1 + (n-1)*abs(incy))`.
incy	`INTEGER`. Specifies the increment for the elements of *y*.
c	`REAL` for `srot` `DOUBLE PRECISION` for `drot` `REAL` for `csrot` `DOUBLE PRECISION` for `zdrot` A scalar.
s	`REAL` for `srot` `DOUBLE PRECISION` for `drot` `REAL` for `csrot` `DOUBLE PRECISION` for `zdrot` A scalar.

Output Parameters

x	Each element is replaced by `c*x + s*y`.
y	Each element is replaced by `c*y - s*x`.

?rotg

Computes the parameters for a Givens rotation.

Syntax

```
call srotg ( a, b, c, s )
call drotg ( a, b, c, s )
call crotg ( a, b, c, s )
call zrotg ( a, b, c, s )
```

Description

Given the cartesian coordinates `(a, b)` of a point `p`, these routines return the parameters *a*, *b*, *c*, and *s* associated with the Givens rotation that zeros the *y*-coordinate of the point.

Input Parameters

a
: `REAL` for `srotg`
`DOUBLE PRECISION` for `drotg`
`COMPLEX` for `crotg`
`DOUBLE COMPLEX` for `zrotg`

Provides the *x*-coordinate of the point `p`.

b
: `REAL` for `srotg`
`DOUBLE PRECISION` for `drotg`
`COMPLEX` for `crotg`
`DOUBLE COMPLEX` for `zrotg`

Provides the *y*-coordinate of the point `p`.

Output Parameters

a
: Contains the parameter *r* associated with the Givens rotation.

b
: Contains the parameter *z* associated with the Givens rotation.

c
: `REAL` for `srotg`
`DOUBLE PRECISION` for `drotg`
`REAL` for `crotg`
`DOUBLE PRECISION` for `zrotg`

Contains the parameter *c* associated with the Givens rotation.

s — `REAL` for `srotg`
`DOUBLE PRECISION` for `drotg`
`COMPLEX` for `crotg`
`DOUBLE COMPLEX` for `zrotg`

Contains the parameter *s* associated with the Givens rotation.

?rotm

Performs rotation of points in the modified plane.

Syntax

```
call srotm ( n, x, incx, y, incy, param )
call drotm ( n, x, incx, y, incy, param )
```

Description

Given two complex vectors *x* and *y*, each vector element of these vectors is replaced as follows:

`x(i) = H*x(i) + H*y(i)`

`y(i) = H*y(i) - H*x(i)`

where:

H is a modified Givens transformation matrix whose values are stored in the *param(2)* through *param(5)* array. See discussion on the *param* argument.

Input Parameters

n — `INTEGER`. Specifies the order of vectors *x* and *y*.

x — `REAL` for `srotm`
`DOUBLE PRECISION` for `drotm`
Array, `DIMENSION` at least `(1 + (n-1)*abs(incx))`.

incx — `INTEGER`. Specifies the increment for the elements of *x*.

y — `REAL` for `srotm`
`DOUBLE PRECISION` for `drotm`
Array, `DIMENSION` at least `(1 + (n-1)*abs(incy))`.

incy — `INTEGER`. Specifies the increment for the elements of *y*.

param — `REAL` for `srotm`
`DOUBLE PRECISION` for `drotm`
Array, `DIMENSION` 5.

The elements of the *param* array are:

param(1) contains a switch, *flag*.
param(2-5) contain *h11*, *h21*, *h12*, and *h22*, respectively, the components of the array *H*.

Depending on the values of *flag*, the components of *H* are set as follows:

$$flag = -1.: H = \begin{bmatrix} h11 & h12 \\ h21 & h22 \end{bmatrix}$$

$$flag = 0.: H = \begin{bmatrix} 1. & h12 \\ h21 & 1. \end{bmatrix}$$

$$flag = 1.: H = \begin{bmatrix} h11 & 1. \\ -1. & h22 \end{bmatrix}$$

$$flag = -2.: H = \begin{bmatrix} 1. & 0. \\ 0. & 1. \end{bmatrix}$$

In the above cases, the matrix entries of 1., -1., and 0. are assumed based on the last three values of *flag* and are not actually loaded into the *param* vector.

Output Parameters

x — Each element is replaced by *h11*x + h12*y*.

y — Each element is replaced by *h21*x + h22*y*.

H — Givens transformation matrix updated.

?rotmg

Computes the modified parameters for a Givens rotation.

Syntax

```
call srotmg ( d1, d2, x1, y1, param )
call drotmg ( d1, d2, x1, y1, param )
```

Description

Given cartesian coordinates (*x1*, *y1*) of an input vector, these routines compute the components of a modified Givens transformation matrix *H* that zeros the *y*-component of the resulting vector:

$$\begin{bmatrix} x \\ 0 \end{bmatrix} = H \begin{bmatrix} x1 \\ y1 \end{bmatrix}$$

Input Parameters

d1 — `REAL` for `srotmg`
`DOUBLE PRECISION` for `drotmg`
Provides the scaling factor for the *x*-coordinate of the input vector (sqrt(*d1*)*x1*).

d2 — `REAL` for `srotmg`
`DOUBLE PRECISION` for `drotmg`
Provides the scaling factor for the *y*-coordinate of the input vector (sqrt(*d2*)*y1*).

x1 — `REAL` for `srotmg`
`DOUBLE PRECISION` for `drotmg`
Provides the *x*-coordinate of the input vector.

y1 — `REAL` for `srotmg`
`DOUBLE PRECISION` for `drotmg`
Provides the *y*-coordinate of the input vector.

Output Parameters

param REAL for srotmg
DOUBLE PRECISION for drotmg
Array, DIMENSION 5.

The elementsof the *param* array are:

param(1) contains a switch, *flag*.
param(2-5) contain *h11*, *h21*, *h12*, and *h22*, respectively, the components of the array *H*.

Depending on the values of *flag*, the components of *H* are set as follows:

$$flag = -1.: H = \begin{bmatrix} h11 & h12 \\ h21 & h22 \end{bmatrix}$$

$$flag = 0.: H = \begin{bmatrix} 1. & h12 \\ h21 & 1. \end{bmatrix}$$

$$flag = 1.: H = \begin{bmatrix} h11 & 1. \\ -1. & h22 \end{bmatrix}$$

$$flag = -2.: H = \begin{bmatrix} 1. & 0. \\ 0. & 1. \end{bmatrix}$$

In the above cases, the matrix entries of 1., -1., and 0. are assumed based on the last three values of *flag* and are not actually loaded into the *param* vector.

?scal

Computes a vector by a scalar product.

Syntax

```
call sscal ( n, a, x, incx )
call dscal ( n, a, x, incx )
call cscal ( n, a, x, incx )
call zscal ( n, a, x, incx )
call csscal ( n, a, x, incx )
```

```
call zdscal ( n, a, x, incx )
```

Description

The ?scal routines perform a vector-vector operation defined as

x = a*x

where:

a is a scalar, *x* is an *n*-element vector.

Input Parameters

n	INTEGER. Specifies the order of vector *x*.
a	REAL for sscal and csscal DOUBLE PRECISION for dscal and zdscal COMPLEX for cscal DOUBLE COMPLEX for zscal Specifies the scalar *a*.
x	REAL for sscal DOUBLE PRECISION for dscal COMPLEX for cscal and csscal DOUBLE COMPLEX for zscal and csscal Array, DIMENSION at least (1 + (*n*-1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for the elements of *x*.

Output Parameters

x	Overwritten by the updated vector x.

?swap

Swaps a vector with another vector.

Syntax

```
call sswap ( n, x, incx, y, incy )
call dswap ( n, x, incx, y, incy )
```

```
call cswap ( n, x, incx, y, incy )
call zswap ( n, x, incx, y, incy )
```

Description

Given the two complex vectors *x* and *y*, the `?swap` routines return vectors *y* and *x* swapped, each replacing the other.

Input Parameters

n	`INTEGER`. Specifies the order of vectors *x* and *y*.
`x`	`REAL` for `sswap` `DOUBLE PRECISION` for `dswap` `COMPLEX` for `cswap` `DOUBLE COMPLEX` for `zswap` Array, `DIMENSION` at least `(1 + (`*n*`-1)*abs(`*incx*`))`.
incx	`INTEGER`. Specifies the increment for the elements of `x`.
y	`REAL` for `sswap` `DOUBLE PRECISION` for `dswap` `COMPLEX` for `cswap` `DOUBLE COMPLEX` for `zswap` Array, `DIMENSION` at least `(1 + (`*n*`-1)*abs(`*incy*`))`.
incy	`INTEGER`. Specifies the increment for the elements of *y*.

Output Parameters

x	Contains the resultant vector *x*.
y	Contains the resultant vector *y*.

i?amax

Finds the element of a vector that has the largest absolute value.

Syntax

```
index = isamax ( n, x, incx )
```

```
index = idamax ( n, x, incx )
index = icamax ( n, x, incx )
index = izamax ( n, x, incx )
```

Description

Given a vector `x`, the `i?amax` functions return the position of the vector element `x(i)` that has the largest absolute value or, for complex flavors, the position of the element with the largest sum |Re `x(i)`| + |Im `x(i)`|.

If *n* is not positive, `0` is returned.

If more than one vector element is found with the same largest absolute value, the index of the first one encountered is returned.

Input Parameters

n	`INTEGER`. Specifies the order of the vector *x*.
x	`REAL` for `isamax` `DOUBLE PRECISION` for `idamax` `COMPLEX` for `icamax` `DOUBLE COMPLEX` for `izamax` Array, `DIMENSION` at least `(1+(n-1)*abs(incx))`.
incx	`INTEGER`. Specifies the increment for the elements of *x*.

Output Parameters

index	`INTEGER`. Contains the position of vector element *x* that has the largest absolute value.

i?amin

Finds the element of a vector that has the smallest absolute value.

Syntax

```
index = isamin ( n, x, incx )
index = idamin ( n, x, incx )
```

```
index = icamin ( n, x, incx )
index = izamin ( n, x, incx )
```

Description

Given a vector `x`, the `i?amin` functions return the position of the vector element `x(i)` that has the smallest absolute value or, for complex flavors, the position of the element with the smallest sum $|\mathrm{Re}\,x(i)| + |\mathrm{Im}\,x(i)|$.

If *n* is not positive, `0` is returned.

If more than one vector element is found with the same smallest absolute value, the index of the first one encountered is returned.

Input Parameters

n	`INTEGER`. On entry, *n* specifies the order of the vector *x*.
x	`REAL` for `isamin` `DOUBLE PRECISION` for `idamin` `COMPLEX` for `icamin` `DOUBLE COMPLEX` for `izamin` Array, `DIMENSION` at least `(1+(n-1)*abs(incx))`.
incx	`INTEGER`. Specifies the increment for the elements of *x*.

Output Parameters

index	`INTEGER`. Contains the position of vector element *x* that has the smallest absolute value.

BLAS Level 2 Routines

This section describes BLAS Level 2 routines, which perform matrix-vector operations. Table 2-2 lists the BLAS Level 2 routine groups and the data types associated with them.

Table 2-2 BLAS Level 2 Routine Groups and Their Data Types

Routine Groups	Data Types	Description
?gbmv	s, d, c, z	Matrix-vector product using a general band matrix
?gemv	s, d, c, z	Matrix-vector product using a general matrix
?ger	s, d	Rank-1 update of a general matrix
?gerc	c, z	Rank-1 update of a conjugated general matrix
?geru	c, z	Rank-1 update of a general matrix, unconjugated
?hbmv	c, z	Matrix-vector product using a Hermitian band matrix
?hemv	c, z	Matrix-vector product using a Hermitian matrix
?her	c, z	Rank-1 update of a Hermitian matrix
?her2	c, z	Rank-2 update of a Hermitian matrix
?hpmv	c, z	Matrix-vector product using a Hermitian packed matrix
?hpr	c, z	Rank-1 update of a Hermitian packed matrix
?hpr2	c, z	Rank-2 update of a Hermitian packed matrix
?sbmv	s, d	Matrix-vector product using symmetric band matrix
?spmv	s, d	Matrix-vector product using a symmetric packed matrix
?spr	s, d	Rank-1 update of a symmetric packed matrix
?spr2	s, d	Rank-2 update of a symmetric packed matrix
?symv	s, d	Matrix-vector product using a symmetric matrix
?syr	s, d	Rank-1 update of a symmetric matrix
?syr2	s, d	Rank-2 update of a symmetric matrix
?tbmv	s, d, c, z	Matrix-vector product using a triangular band matrix
?tbsv	s, d, c, z	Linear solution of a triangular band matrix

Table 2-2 BLAS Level 2 Routine Groups and Their Data Types (continued)

Routine Groups	Data Types	Description
?tpmv	s, d, c, z	Matrix-vector product using a triangular packed matrix
?tpsv	s, d, c, z	Linear solution of a triangular packed matrix
?trmv	s, d, c, z	Matrix-vector product using a triangular matrix
?trsv	s, d, c, z	Linear solution of a triangular matrix

?gbmv

Computes a matrix-vector product using a general band matrix

Syntax

```
call sgbmv ( trans, m, n, kl, ku, alpha, a, lda, x, inxc, beta, y, incy )
call dgbmv ( trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy )
call cgbmv ( trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy )
call zgbmv ( trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy )
```

Description

The ?gbmv routines perform a matrix-vector operation defined as

y := *alpha***a***x* + *beta***y*

or

y := *alpha***a*'**x* + *beta***y*,

or

y := *alpha**conjg(*a*')**x* + *beta***y*,

where:

alpha and *beta* are scalars

x and *y* are vectors

a is an *m* by *n* band matrix, with *kl* sub-diagonals and *ku* super-diagonals.

Input Parameters

trans — CHARACTER*1. Specifies the operation to be performed, as follows:

trans value	Operation to be Performed
N or n	$y := alpha*a*x + beta*y$
T or t	$y := alpha*a'*x + beta*y$
C or c	$y := alpha*conjg(a')*x + beta*y$

m — INTEGER. Specifies the number of rows of the matrix *a*. The value of *m* must be at least zero.

n — INTEGER. Specifies the number of columns of the matrix *a*. The value of *n* must be at least zero.

kl — INTEGER. Specifies the number of sub-diagonals of the matrix *a*. The value of *kl* must satisfy $0 \le kl$.

ku — INTEGER. Specifies the number of super-diagonals of the matrix *a*. The value of *ku* must satisfy $0 \le ku$.

alpha — REAL for sgbmv
DOUBLE PRECISION for dgbmv
COMPLEX for cgbmv
DOUBLE COMPLEX for zgbmv

Specifies the scalar *alpha*.

a — REAL for sgbmv
DOUBLE PRECISION for dgbmv
COMPLEX for cgbmv
DOUBLE COMPLEX for zgbmv

Array, DIMENSION (*lda*, *n*). Before entry, the leading $(kl + ku + 1)$ by *n* part of the array *a* must contain the matrix of coefficients. This matrix must be supplied column-by-column, with the leading diagonal of the matrix in row $(ku + 1)$ of the array, the first super-diagonal starting at position 2 in row *ku*, the first sub-diagonal starting at position 1 in row $(ku + 2)$, and so on. Elements in the array *a* that do not correspond to elements in the band matrix (such as the top left *ku* by *ku* triangle) are not referenced.

The following program segment transfers a band matrix from conventional full matrix storage to band storage:

```
do 20, j = 1, n
 k = ku + 1 - j
    do 10, i = max(1, j-ku), min(m, j+kl)
   a(k+i, j) = matrix(i,j)
 10 continue
20 continue
```

lda — `INTEGER`. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least `(kl + ku + 1)`.

x — `REAL` for `sgbmv`
`DOUBLE PRECISION` for `dgbmv`
`COMPLEX` for `cgbmv`
`DOUBLE COMPLEX` for `zgbmv`

Array, `DIMENSION` at least `(1 + (n - 1)*abs(incx))` when *trans* = `'N'` or `'n'` and at least `(1 + (m - 1)*abs(incx))` otherwise. Before entry, the incremented array *x* must contain the vector *x*.

incx — `INTEGER`. Specifies the increment for the elements of *x*. *incx* must not be zero.

beta — `REAL` for `sgbmv`
`DOUBLE PRECISION` for `dgbmv`
`COMPLEX` for `cgbmv`
`DOUBLE COMPLEX` for `zgbmv`

Specifies the scalar beta. When *beta* is supplied as zero, then *y* need not be set on input.

y — `REAL` for `sgbmv`
`DOUBLE PRECISION` for `dgbmv`
`COMPLEX` for `cgbmv`
`DOUBLE COMPLEX` for `zgbmv`

Array, `DIMENSION` at least `(1 + (m - 1)*abs(incy))` when *trans* = `'N'` or `'n'` and at least
`(1 + (n - 1)*abs(incy))` otherwise. Before entry, the incremented array *y* must contain the vector *y*.

incy — `INTEGER`. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y — Overwritten by the updated vector *y*.

?gemv

Computes a matrix-vector product using a general matrix

Syntax

```
call sgemv ( trans, m, n, alpha, a, lda, x, incx, beta, y, incy )
call dgemv ( trans, m, n, alpha, a, lda, x, incx, beta, y, incy )
call cgemv ( trans, m, n, alpha, a, lda, x, incx, beta, y, incy )
call zgemv ( trans, m, n, alpha, a, lda, x, incx, beta, y, incy )
```

Description

The `?gemv` routines perform a matrix-vector operation defined as

`y := alpha*a*x + beta*y,`

or

`y := alpha*a'*x + beta*y,`

or

`y := alpha*conjg(a')*x + beta*y,`

where:

alpha and *beta* are scalars

x and *y* are vectors

a is an *m* by *n* matrix.

Input Parameters

trans CHARACTER*1. Specifies the operation to be performed, as follows:

trans value	Operation to be Performed
N or n	`y:= alpha*a*x + beta*y`
T or t	`y:= alpha*a'*x + beta*y`
C or c	`y:= alpha*conjg(a')*x +beta*y`

m	INTEGER. Specifies the number of rows of the matrix *a*. *m* must be at least zero.
n	INTEGER. Specifies the number of columns of the matrix *a*. The value of *n* must be at least zero.
alpha	REAL for sgemv DOUBLE PRECISION for dgemv COMPLEX for cgemv DOUBLE COMPLEX for zgemv Specifies the scalar alpha.
a	REAL for sgemv DOUBLE PRECISION for dgemv COMPLEX for cgemv DOUBLE COMPLEX for zgemv Array, DIMENSION (*lda*, *n*). Before entry, the leading *m* by *n* part of the array *a* must contain the matrix of coefficients.
lda	INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *m*).
x	REAL for sgemv DOUBLE PRECISION for dgemv COMPLEX for cgemv DOUBLE COMPLEX for zgemv Array, DIMENSION at least (1+(*n*-1)*abs(*incx*)) when *trans* = 'N' or 'n' and at least (1+(*m* - 1)*abs(*incx*)) otherwise. Before entry, the incremented array *x* must contain the vector *x*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.
beta	REAL for sgemv DOUBLE PRECISION for dgemv COMPLEX for cgemv DOUBLE COMPLEX for zgemv Specifies the scalar *beta*. When *beta* is supplied as zero, then *y* need not be set on input.

y — `REAL` for `sgemv`
`DOUBLE PRECISION` for `dgemv`
`COMPLEX` for `cgemv`
`DOUBLE COMPLEX` for `zgemv`

Array, `DIMENSION` at least `(1 + (`*m*` - 1)*abs(`*incy*`))` when *trans* = `'N'` or `'n'` and at least `(1 + (`*n*` - 1)*abs(`*incy*`))` otherwise. Before entry with *beta* non-zero, the incremented array *y* must contain the vector *y*.

incy — `INTEGER`. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y — Overwritten by the updated vector *y*.

?ger

Performs a rank-1 update of a general matrix.

Syntax

```
call sger ( m, n, alpha, x, incx, y, incy, a, lda )
call dger ( m, n, alpha, x, incx, y, incy, a, lda )
```

Description

The `?ger` routines perform a matrix-vector operation defined as

a := *alpha***x***y*' + *a*,

where:

alpha is a scalar

x is an *m*-element vector

y is an *n*-element vector

a is an *m* by *n* matrix.

Input Parameters

m	`INTEGER`. Specifies the number of rows of the matrix *a*. The value of *m* must be at least zero.
n	`INTEGER`. Specifies the number of columns of the matrix *a*. The value of *n* must be at least zero.
alpha	`REAL` for `sger` `DOUBLE PRECISION` for `dger` Specifies the scalar *alpha*.
x	`REAL` for `sger` `DOUBLE PRECISION` for `dger` Array, `DIMENSION` at least `(1 + (`*m* `- 1)*abs(`*incx*`))`. Before entry, the incremented array *x* must contain the *m*-element vector *x*.
incx	`INTEGER`. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.
y	`REAL` for `sger` `DOUBLE PRECISION` for `dger` Array, `DIMENSION` at least `(1 + (`*n* `- 1)*abs(`*incy*`))`. Before entry, the incremented array *y* must contain the *n*-element vector *y*.
incy	`INTEGER`. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.
a	`REAL` for `sger` `DOUBLE PRECISION` for `dger` Array, `DIMENSION (`*lda*, *n*`)`. Before entry, the leading *m* by *n* part of the array *a* must contain the matrix of coefficients.
lda	`INTEGER`. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least `max(1,` *m*`)`.

Output Parameters

a	Overwritten by the updated matrix.

?gerc

Performs a rank-1 update (conjugated) of a general matrix.

Syntax

```
call cgerc ( m, n, alpha, x, incx, y, incy, a, lda )
call zgerc ( m, n, alpha, x, incx, y, incy, a, lda )
```

Description

The `?gerc` routines perform a matrix-vector operation defined as

`a := alpha*x*conjg(y') + a,`

where:

alpha is a scalar

x is an *m*-element vector

y is an *n*-element vector

a is an *m* by *n* matrix.

Input Parameters

m	`INTEGER`. Specifies the number of rows of the matrix *a*. The value of *m* must be at least zero.
n	`INTEGER`. Specifies the number of columns of the matrix *a*. The value of *n* must be at least zero.
alpha	`SINGLE PRECISION COMPLEX` for `cgerc` `DOUBLE PRECISION COMPLEX` for `zgerc` Specifies the scalar *alpha*.
x	`SINGLE PRECISION COMPLEX` for `cgerc` `DOUBLE PRECISION COMPLEX` for `zgerc` Array, `DIMENSION` at least `(1 + (`*m* `- 1)*abs(`*incx*`))`. Before entry, the incremented array *x* must contain the *m*-element vector *x*.

incx — `INTEGER`. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

y — `COMPLEX` for `cgerc`
`DOUBLE COMPLEX` for `zgerc`

Array, `DIMENSION` at least `(1 + (n - 1)*abs(incy))`. Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy — `INTEGER`. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

a — `COMPLEX` for `cgerc`
`DOUBLE COMPLEX` for `zgerc`

Array, `DIMENSION (lda, n)`. Before entry, the leading *m* by *n* part of the array *a* must contain the matrix of coefficients.

lda — `INTEGER`. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least `max(1, m)`.

Output Parameters

a — Overwritten by the updated matrix.

?geru

Performs a rank-1 update (unconjugated) of a general matrix.

Syntax

```
call cgeru ( m, n, alpha, x, incx, y, incy, a, lda )
call zgeru ( m, n, alpha, x, incx, y, incy, a, lda )
```

Description

The `?geru` routines perform a matrix-vector operation defined as

`a:= alpha*x*y' + a,`

where:

alpha is a scalar

x is an *m*-element vector

y is an *n*-element vector

a is an *m* by *n* matrix.

Input Parameters

m	`INTEGER`. Specifies the number of rows of the matrix *a*. The value of *m* must be at least zero.
n	`INTEGER`. Specifies the number of columns of the matrix *a*. The value of *n* must be at least zero.
alpha	`COMPLEX` for `cgeru` `DOUBLE COMPLEX` for `zgeru` Specifies the scalar *alpha*.
x	`COMPLEX` for `cgeru` `DOUBLE COMPLEX` for `zgeru` Array, `DIMENSION` at least `(1 + (`*m*` - 1)*abs(`*incx*`))`. Before entry, the incremented array *x* must contain the *m*-element vector *x*.
incx	`INTEGER`. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.
y	`COMPLEX` for `cgeru` `DOUBLE COMPLEX` for `zgeru` Array, `DIMENSION` at least `(1 + (`*n*` - 1)*abs(`*incy*`))`. Before entry, the incremented array *y* must contain the *n*-element vector *y*.
incy	`INTEGER`. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.
a	`COMPLEX` for `cgeru` `DOUBLE COMPLEX` for `zgeru` Array, `DIMENSION (`*lda*`, `*n*`)`. Before entry, the leading *m* by *n* part of the array *a* must contain the matrix of coefficients.
lda	`INTEGER`. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least `max(1, `*m*`)`.

Output Parameters

a	Overwritten by the updated matrix.

?hbmv

Computes a matrix-vector product using a Hermitian band matrix.

Syntax

```
call chbmv ( uplo, n, k, alpha, a, lda, x, incx, beta, y, incy )
call zhbmv ( uplo, n, k, alpha, a, lda, x, incx, beta, y, incy )
```

Description

The `?hbmv` routines perform a matrix-vector operation defined as

*y := alpha*a*x + beta*y,*

where:

alpha and *beta* are scalars

x and *y* are *n*-element vectors

a is an *n* by *n* Hermitian band matrix, with *k* super-diagonals.

Input Parameters

uplo `CHARACTER*1`. Specifies whether the upper or lower triangular part of the band matrix *a* is being supplied, as follows:

uplo value	Part of Matrix *a* Supplied
U or u	The upper triangular part of matrix *a* is being supplied.
L or l	The lower triangular part of matrix *a* is being supplied.

n `INTEGER`. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

k `INTEGER`. Specifies the number of super-diagonals of the matrix *a*. The value of *k* must satisfy $0 \le k$.

alpha `COMPLEX` for `chbmv`
`DOUBLE COMPLEX` for `zhbmv`

Specifies the scalar *alpha*.

a — COMPLEX for chbmv
DOUBLE COMPLEX for zhbmv

Array, DIMENSION (*lda, n*). Before entry with *uplo* = 'U' or 'u', the leading (*k* + 1) by *n* part of the array *a* must contain the upper triangular band part of the Hermitian matrix. The matrix must be supplied column-by-column, with the leading diagonal of the matrix in row (*k* + 1) of the array, the first super-diagonal starting at position 2 in row *k*, and so on. The top left *k* by *k* triangle of the array *a* is not referenced.

The following program segment transfers the upper triangular part of a Hermitian band matrix from conventional full matrix storage to band storage:

```
do 20, j = 1, n
 m = k + 1 - j
 do 10, i = max(1, j - k), j
  a(m + i, j) = matrix(i, j)
 10 continue
20 continue
```

Before entry with *uplo* = 'L' or 'l', the leading (*k* + 1) by *n* part of the array *a* must contain the lower triangular band part of the Hermitian matrix, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right *k* by *k* triangle of the array *a* is not referenced.

The following program segment transfers the lower triangular part of a Hermitian band matrix from conventional full matrix storage to band storage:

```
do 20, j = 1, n
 m = 1 - j
 do 10, i = j, min( n, j + k )
  a( m + i, j ) = matrix( i, j )
 10 continue
20 continue
```

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

lda — INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least (*k* + 1).

x — COMPLEX for chbmv
DOUBLE COMPLEX for zhbmv

Array, `DIMENSION` at least `(1 + (n - 1)*abs(incx))`. Before entry, the incremented array *x* must contain the vector *x*.

incx — `INTEGER`. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

beta — `COMPLEX` for `chbmv`
`DOUBLE COMPLEX` for `zhbmv`

Specifies the scalar *beta*.

y — `COMPLEX` for `chbmv`
`DOUBLE COMPLEX` for `zhbmv`

Array, `DIMENSION` at least `(1 + (n - 1)*abs(incy))`. Before entry, the incremented array *y* must contain the vector *y*.

incy — `INTEGER`. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y — Overwritten by the updated vector *y*.

?hemv

Computes a matrix-vector product using a Hermitian matrix.

Syntax

```
call chemv ( uplo, n, alpha, a, lda, x, incx, beta, y, incy )
call zhemv ( uplo, n, alpha, a, lda, x, incx, beta, y, incy )
```

Description

The `?hemv` routines perform a matrix-vector operation defined as

*y := alpha*a*x + beta*y,*

where:

alpha and *beta* are scalars

x and *y* are *n*-element vectors

a is an *n* by *n* Hermitian matrix.

Input Parameters

uplo — `CHARACTER*1`. Specifies whether the upper or lower triangular part of the array *a* is to be referenced, as follows:

uplo value	Part of Array *a* To Be Referenced
U or u	The upper triangular part of array *a* is to be referenced.
L or l	The lower triangular part of array *a* is to be referenced.

n — `INTEGER`. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

alpha — `COMPLEX` for `chemv`
`DOUBLE COMPLEX` for `zhemv`

Specifies the scalar *alpha*.

a — `COMPLEX` for `chemv`
`DOUBLE COMPLEX` for `zhemv`

Array, `DIMENSION` (*lda*, *n*). Before entry with *uplo* = 'U' or 'u', the leading *n* by *n* upper triangular part of the array *a* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *n* by *n* lower triangular part of the array *a* must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of *a* is not referenced.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

lda — `INTEGER`. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least `max(1, n)`.

x — `COMPLEX` for `chemv`
`DOUBLE COMPLEX` for `zhemv`

Array, `DIMENSION` at least `(1 + (n - 1)*abs(incx))`. Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

beta COMPLEX for chemv
DOUBLE COMPLEX for zhemv

Specifies the scalar *beta*. When *beta* is supplied as zero then *y* need not be set on input.

y COMPLEX for chemv
DOUBLE COMPLEX for zhemv

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y Overwritten by the updated vector *y*.

?her

Performs a rank-1 update of a Hermitian matrix.

Syntax

```
call cher ( uplo, n, alpha, x, incx, a, lda )
call zher ( uplo, n, alpha, x, incx, a, lda )
```

Description

The ?her routines perform a matrix-vector operation defined as

a := *alpha***x**conjg(*x*') + *a*,

where:

alpha is a real scalar

x is an *n*-element vector

a is an *n* by *n* Hermitian matrix.

Input Parameters

uplo — `CHARACTER*1`. Specifies whether the upper or lower triangular part of the array `a` is to be referenced, as follows:

uplo value	Part of Array *a* To Be Referenced
`U` or `u`	The upper triangular part of array *a* is to be referenced.
`L` or `l`	The lower triangular part of array *a* is to be referenced.

n — `INTEGER`. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

alpha — `REAL` for `cher`
`DOUBLE PRECISION` for `zher`

Specifies the scalar *alpha*.

x — `COMPLEX` for `cher`
`DOUBLE COMPLEX` for `zher`

Array, dimension at least `(1 + (`*n*` - 1)*abs(`*incx*`))`. Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx — `INTEGER`. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

a — `COMPLEX` for `cher`
`DOUBLE COMPLEX` for `zher`

Array, `DIMENSION (`*lda*`, `*n*`)`. Before entry with *uplo* = `'U'` or `'u'`, the leading *n* by *n* upper triangular part of the array *a* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *a* is not referenced.

Before entry with *uplo* = `'L'` or `'l'`, the leading *n* by *n* lower triangular part of the array *a* must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of *a* is not referenced.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

lda — `INTEGER`. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least `max(1, `*n*`)`.

Output Parameters

a With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements are set to zero.

?her2

Performs a rank-2 update of a Hermitian matrix.

Syntax

```
call cher2 ( uplo, n, alpha, x, incx, y, incy, a, lda )
call zher2 ( uplo, n, alpha, x, incx, y, incy, a, lda )
```

Description

The ?her2 routines perform a matrix-vector operation defined as

```
a := alpha*x*conjg(y') + conjg(alpha)*y*conjg(x') + a,
```

where:

alpha is a scalar

x and *y* are *n*-element vectors

a is an *n* by *n* Hermitian matrix.

Input Parameters

uplo CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is to be referenced, as follows:

uplo value	Part of Array *a* To Be Referenced
U or u	The upper triangular part of array *a* is to be referenced.
L or l	The lower triangular part of array *a* is to be referenced.

n	`INTEGER`. Specifies the order of the matrix *a*. The value of *n* must be at least zero.
alpha	`COMPLEX` for `cher2` `DOUBLE COMPLEX` for `zher2` Specifies the scalar *alpha*.
x	`COMPLEX` for `cher2` `DOUBLE COMPLEX` for `zher2` Array, `DIMENSION` at least `(1 + (`*n* `- 1)*abs(`*incx*`))`. Before entry, the incremented array *x* must contain the *n*-element vector *x*.
incx	`INTEGER`. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.
y	`COMPLEX` for `cher2` `DOUBLE COMPLEX` for `zher2` Array, `DIMENSION` at least `(1 + (`*n* `- 1)*abs(`*incy*`))`. Before entry, the incremented array *y* must contain the *n*-element vector *y*.
incy	`INTEGER`. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.
a	`COMPLEX` for `cher2` `DOUBLE COMPLEX` for `zher2` Array, `DIMENSION (`*lda, n*`)`. Before entry with *uplo* = `'U'` or `'u'`, the leading *n* by *n* upper triangular part of the array *a* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = `'L'` or `'l'`, the leading *n* by *n* lower triangular part of the array *a* must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of *a* is not referenced. The imaginary parts of the diagonal elements need not be set and are assumed to be zero.
lda	`INTEGER`. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least `max(1,` *n*`)`.

Output Parameters

a	With *uplo* = `'U'` or `'u'`, the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements are set to zero.

?hpmv

Computes a matrix-vector product using a Hermitian packed matrix.

Syntax

```
call chpmv ( uplo, n, alpha, ap, x, incx, beta, y, incy )
call zhpmv ( uplo, n, alpha, ap, x, incx, beta, y, incy )
```

Description

The ?hpmv routines perform a matrix-vector operation defined as

*y := alpha*a*x + beta*y,*

where:

alpha and *beta* are scalars

x and *y* are *n*-element vectors

a is an *n* by *n* Hermitian matrix, supplied in packed form.

Input Parameters

uplo CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *a* is supplied in the packed array *ap*, as follows:

uplo value	Part of Matrix *a* Supplied
U or u	The upper triangular part of matrix *a* is supplied in *ap*.
L or l	The lower triangular part of matrix a is supplied in *ap*.

n INTEGER. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

alpha — COMPLEX for chpmv
DOUBLE COMPLEX for zhpmv

Specifies the scalar *alpha*.

ap — COMPLEX for chpmv
DOUBLE COMPLEX for zhpmv

Array, DIMENSION at least ((*n**(*n* + 1))/2). Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(1, 2) and *a*(2, 2) respectively, and so on. Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), ap(2) and *ap*(3) contain *a*(2, 1) and *a*(3, 1) respectively, and so on.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

x — COMPLEX for chpmv
DOUBLE PRECISION COMPLEX for zhpmv

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

beta — COMPLEX for chpmv
DOUBLE COMPLEX for zhpmv

Specifies the scalar *beta*. When *beta* is supplied as zero then *y* need not be set on input.

y — COMPLEX for chpmv
DOUBLE COMPLEX for zhpmv

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy — INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y — Overwritten by the updated vector *y*

?hpr

Performs a rank-1 update of a Hermitian packed matrix.

Syntax

```
call chpr ( uplo, n, alpha, x, incx, ap )
call zhpr ( uplo, n, alpha, x, incx, ap )
```

Description

The?hpr routines perform a matrix-vector operation defined as

a := alpha*x*conjg(x') + a,

where:

alpha is a real scalar

x is an *n*-element vector

a is an *n* by *n* Hermitian matrix, supplied in packed form.

Input Parameters

uplo CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *a* is supplied in the packed array *ap*, as follows:

uplo value	Part of Matrix *a* Supplied
U or u	The upper triangular part of matrix *a* is supplied in *ap*.
L or l	The lower triangular part of matrix *a* is supplied in *ap*.

n INTEGER. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

alpha REAL for chpr
DOUBLE PRECISION for zhpr

Specifies the scalar *alpha*.

x COMPLEX for chpr
DOUBLE COMPLEX for zhpr

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx INTEGER. Specifies the increment for the elements of *x*. *incx* must not be zero.

ap COMPLEX for chpr
DOUBLE COMPLEX for zhpr

Array, DIMENSION at least ((*n**(*n* + 1))/2). Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(1, 2) and *a*(2, 2) respectively, and so on.

Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(2, 1) and *a*(3, 1) respectively, and so on.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

Output Parameters

ap With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements are set to zero.

?hpr2

Performs a rank-2 update of a Hermitian packed matrix.

Syntax

```
call chpr2 ( uplo, n, alpha, x, incx, y, incy, ap )
```

`call zhpr2 ( uplo, n, alpha, x, incx, y, incy, ap )`

Description

The?`hpr2` routines perform a matrix-vector operation defined as

`a := alpha*x*conjg(y') + conjg(alpha)*y*conjg(x') + a,`

where:

alpha is a scalar

x and *y* are *n*-element vectors

a is an *n* by *n* Hermitian matrix, supplied in packed form.

Input Parameters

uplo — `CHARACTER*1`. Specifies whether the upper or lower triangular part of the matrix *a* is supplied in the packed array *ap*, as follows

uplo **value**	**Part of Matrix** *a* **Supplied**
U or u	The upper triangular part of matrix *a* is supplied in *ap*.
L or l	The lower triangular part of matrix *a* is supplied in *ap*.

n — `INTEGER`. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

alpha — `COMPLEX` for `chpr2`
`DOUBLE COMPLEX` for `zhpr2`

Specifies the scalar *alpha*.

x — `COMPLEX` for `chpr2`
`DOUBLE COMPLEX` for `zhpr2`

Array, dimension at least `(1 + (n - 1)*abs(incx))`. Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx — `INTEGER`. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

y — `COMPLEX` for `chpr2`
`DOUBLE COMPLEX` for `zhpr2`

Array, `DIMENSION` at least `(1 + (n - 1)*abs(incy))`. Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy — `INTEGER`. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

ap — `COMPLEX` for `chpr2`
`DOUBLE COMPLEX` for `zhpr2`

Array, `DIMENSION` at least `((n*(n + 1))/2)`. Before entry with *uplo* = `'U'` or `'u'`, the array *ap* must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*`(1)` contains *a*`(1, 1)`, *ap*`(2)` and *ap*`(3)` contain *a*`(1, 2)` and *a*`(2, 2)` respectively, and so on.

Before entry with *uplo* = `'L'` or `'l'`, the array *ap* must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*`(1)` contains *a*`(1, 1)`, *ap*`(2)` and *ap*`(3)` contain *a*`(2, 1)` and *a*`(3, 1)` respectively, and so on.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

Output Parameters

ap — With *uplo* = `'U'` or `'u'`, overwritten by the upper triangular part of the updated matrix.

With *uplo* = `'L'` or `'l'`, overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements need are set to zero.

?sbmv

Computes a matrix-vector product using a symmetric band matrix.

Syntax

```
call ssbmv ( uplo, n, k, alpha, a, lda, x, incx, beta, y, incy )
call dsbmv ( uplo, n, k, alpha, a, lda, x, incx, beta, y, incy )
```

Description

The ?sbmv routines perform a matrix-vector operation defined as

*y := alpha*a*x + beta*y,*

where:

alpha and *beta* are scalars

x and *y* are *n*-element vectors

a is an *n* by *n* symmetric band matrix, with *k* super-diagonals.

Input Parameters

uplo — CHARACTER*1. Specifies whether the upper or lower triangular part of the band matrix *a* is being supplied, as follows:

uplo value	Part of Matrix *a* Supplied
U or u	The upper triangular part of matrix *a* is supplied.
L or l	The lower triangular part of matrix *a* is supplied.

n — INTEGER. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

k — INTEGER. Specifies the number of super-diagonals of the matrix *a*. The value of *k* must satisfy $0 \le k$.

alpha — REAL for ssbmv
DOUBLE PRECISION for dsbmv

Specifies the scalar *alpha*.

a — REAL for ssbmv
DOUBLE PRECISION for dsbmv

Array, DIMENSION (*lda, n*). Before entry with *uplo* = 'U' or 'u', the leading (*k* + 1) by *n* part of the array *a* must contain the upper triangular band part of the symmetric matrix, supplied column-by-column, with the leading diagonal of the matrix in row (*k* + 1) of the array, the first super-diagonal starting at position 2 in row *k*, and so on. The top left *k* by *k* triangle of the array *a* is not referenced.

The following program segment transfers the upper triangular part of a symmetric band matrix from conventional full matrix storage to band storage:

```
 do 20, j = 1, n
  m = k + 1 - j
  do 10, i = max( 1, j - k ), j
   a( m + i, j ) = matrix( i, j )
   10 continue
 20 continue
```

Before entry with *uplo* = 'L' or 'l', the leading (*k* + 1) by *n* part of the array *a* must contain the lower triangular band part of the symmetric matrix, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right *k* by *k* triangle of the array *a* is not referenced.

The following program segment transfers the lower triangular part of a symmetric band matrix from conventional full matrix storage to band storage:

```
do 20, j = 1, n
 m = 1 - j
 do 10, i = j, min( n, j + k )
  a( m + i, j ) = matrix( i, j )
 10 continue
20 continue
```

lda — INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least (*k* + 1).

x — REAL for ssbmv
DOUBLE PRECISION for dsbmv

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

beta — REAL for ssbmv
DOUBLE PRECISION for dsbmv

Specifies the scalar *beta*.

y — REAL for ssbmv
DOUBLE PRECISION for dsbmv

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the vector *y*.

incy INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y Overwritten by the updated vector *y*.

?spmv

Computes a matrix-vector product using a symmetric packed matrix.

Syntax

```
call sspmv ( uplo, n, alpha, ap, x, incx, beta, y, incy )
call dspmv ( uplo, n, alpha, ap, x, incx, beta, y, incy )
```

Description

The `?spmv` routines perform a matrix-vector operation defined as

*y := alpha*a*x + beta*y,*

where:

alpha and *beta* are scalars

x and *y* are *n*-element vectors

a is an *n* by *n* symmetric matrix, supplied in packed form.

Input Parameters

uplo CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *a* is supplied in the packed array *ap*, as follows:

uplo value	Part of Matrix *a* Supplied
U or u	The upper triangular part of matrix *a* is supplied in *ap*.
L or l	The lower triangular part of matrix *a* is supplied in *ap*.

n — INTEGER. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

alpha — REAL for sspmv
DOUBLE PRECISION for dspmv

Specifies the scalar *alpha*.

ap — REAL for sspmv
DOUBLE PRECISION for dspmv

Array, DIMENSION at least ((n*(n + 1))/2). Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(1, 2) and *a*(2, 2) respectively, and so on. Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(2, 1) and *a*(3, 1) respectively, and so on.

x — REAL for sspmv
DOUBLE PRECISION for dspmv

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array x must contain the *n*-element vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

beta — REAL for sspmv
DOUBLE PRECISION for dspmv

Specifies the scalar *beta*. When *beta* is supplied as zero, then *y* need not be set on input.

y — REAL for sspmv
DOUBLE PRECISION for dspmv

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy — INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y — Overwritten by the updated vector *y*.

?spr

Performs a rank-1 update of a symmetric packed matrix.

Syntax

```
call sspr( uplo, n, alpha, x, incx, ap )
call dspr( uplo, n, alpha, x, incx, ap )
```

Description

The `?spr` routines perform a matrix-vector operation defined as

a:= *alpha***x***x*' + *a*,

where:

alpha is a real scalar

x is an *n*-element vector

a is an *n* by *n* symmetric matrix, supplied in packed form.

Input Parameters

uplo — `CHARACTER*1`. Specifies whether the upper or lower triangular part of the matrix *a* is supplied in the packed array *ap*, as follows:

uplo value	Part of Matrix *a* Supplied
U or u	The upper triangular part of matrix *a* is supplied in *ap*.
L or l	The lower triangular part of matrix *a* is supplied in *ap*.

n — `INTEGER`. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

alpha — `REAL` for `sspr`
`DOUBLE PRECISION` for `dspr`

Specifies the scalar *alpha*.

x — REAL for sspr
DOUBLE PRECISION for dspr

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

ap — REAL for sspr
DOUBLE PRECISION for dspr

Array, DIMENSION at least ((*n**(*n* + 1))/2). Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a*(1, 2) and *a*(2,2) respectively, and so on.

Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2)and *ap*(3)contain *a*(2,1) and *a*(3,1) respectively, and so on.

Output Parameters

ap — With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.

?spr2

Performs a rank-2 update of a symmetric packed matrix.

Syntax

```
call sspr2( uplo, n, alpha, x, incx, y, incy, ap )
call dspr2( uplo, n, alpha, x, incx, y, incy, ap )
```

Description

The ?spr2 routines perform a matrix-vector operation defined as

a:= *alpha***x***y*' + *alpha***y***x*' + *a*,

where:

alpha is a scalar

x and *y* are *n*-element vectors

a is an *n* by *n* symmetric matrix, supplied in packed form.

Input Parameters

uplo
: CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *a* is supplied in the packed array *ap*, as follows:

uplo value	Part of Matrix *a* Supplied
U or u	The upper triangular part of matrix *a* is supplied in *ap*.
L or l	The lower triangular part of matrix *a* is supplied in *ap*.

n
: INTEGER. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

alpha
: REAL for sspr2
DOUBLE PRECISION for dspr2

Specifies the scalar *alpha*.

x
: REAL for sspr2
DOUBLE PRECISION for dspr2

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx
: INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

y
: REAL for sspr2
DOUBLE PRECISION for dspr2

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy — `INTEGER`. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

ap — `REAL` for `sspr2`
`DOUBLE PRECISION` for `dspr2`

Array, `DIMENSION` at least `((`*n*`*(`*n*` + 1))/2)`. Before entry with *uplo* = `'U'` or `'u'`, the array *ap* must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*`(1)` contains *a*`(1,1)`, *ap*`(2)` and *ap*`(3)` contain *a*`(1,2)` and *a*`(2,2)` respectively, and so on.

Before entry with *uplo* = `'L'` or `'l'`, the array *ap* must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*`(1)` contains *a*`(1,1)`, *ap*`(2)` and *ap*`(3)` contain *a*`(2,1)` and *a*`(3,1)` respectively, and so on.

Output Parameters

ap — With *uplo* = `'U'` or `'u'`, overwritten by the upper triangular part of the updated matrix.

With *uplo* = `'L'` or `'l'`, overwritten by the lower triangular part of the updated matrix.

?symv

Computes a matrix-vector product for a symmetric matrix.

Syntax

```
call ssymv ( uplo, n, alpha, a, lda, x, incx, beta, y, incy )
call dsymv ( uplo, n, alpha, a, lda, x, incx, beta, y, incy )
```

Description

The `?symv` routines perform a matrix-vector operation defined as

y := *alpha***a***x* + *beta***y*,

where:

alpha and *beta* are scalars

x and *y* are *n*-element vectors

a is an *n* by *n* symmetric matrix.

Input Parameters

uplo CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is to be referenced, as follows:

uplo value	Part of Array *a* To Be Referenced
U or u	The upper triangular part of array *a* is to be referenced.
L or l	The lower triangular part of array *a* is to be referenced.

n INTEGER. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

alpha REAL for ssymv
DOUBLE PRECISION for dsymv

Specifies the scalar *alpha*.

a REAL for ssymv
DOUBLE PRECISION for dsymv

Array, DIMENSION (*lda*, *n*). Before entry with *uplo* = 'U' or 'u', the leading *n* by *n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *n* by *n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.

lda INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1,*n*).

x REAL for ssymv
DOUBLE PRECISION for dsymv

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

beta — REAL for ssymv
DOUBLE PRECISION for dsymv

Specifies the scalar *beta*. When *beta* is supplied as zero, then *y* need not be set on input.

y — REAL for ssymv
DOUBLE PRECISION for dsymv

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy — INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y — Overwritten by the updated vector *y*.

?syr

Performs a rank-1 update of a symmetric matrix.

Syntax

```
call ssyr( uplo, n, alpha, x, incx, a, lda )
call dsyr( uplo, n, alpha, x, incx, a, lda )
```

Description

The ?syr routines perform a matrix-vector operation defined as

a := *alpha***x***x*' + *a*,

where:

alpha is a real scalar

x is an *n*-element vector

a is an *n* by *n* symmetric matrix.

Input Parameters

uplo CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is to be referenced, as follows:

uplo value	Part of Array *a* To Be Referenced
U or u	The upper triangular part of array *a* is to be referenced.
L or l	The lower triangular part of array *a* is to be referenced.

n INTEGER. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

alpha REAL for ssyr
DOUBLE PRECISION for dsyr

Specifies the scalar *alpha*.

x REAL for ssyr
DOUBLE PRECISION for dsyr

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array x must contain the *n*-element vector *x*.

incx INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

a REAL for ssyr
DOUBLE PRECISION for dsyr

Array, DIMENSION (*lda*, *n*). Before entry with *uplo* = 'U' or 'u', the leading *n* by *n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced.

Before entry with *uplo* = 'L' or 'l', the leading *n* by *n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.

lda INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1,*n*).

Output Parameters

a With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.

?syr2

Performs a rank-2 update of symmetric matrix.

Syntax

```
call ssyr2( uplo, n, alpha, x, incx, y, incy, a, lda )
call dsyr2( uplo, n, alpha, x, incx, y, incy, a, lda )
```

Description

The ?syr2 routines perform a matrix-vector operation defined as

a := *alpha***x***y*' + *alpha***y***x*' + *a*,

where:

alpha is a scalar

x and *y* are *n*-element vectors

a is an *n* by *n* symmetric matrix.

Input Parameters

uplo CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is to be referenced, as follows:

uplo value	Part of Array *a* To Be Referenced
U or u	The upper triangular part of array *a* is to be referenced.
L or l	The lower triangular part of array *a* is to be referenced.

n	`INTEGER`. Specifies the order of the matrix *a*. The value of *n* must be at least zero.
alpha	`REAL` for `ssyr2` `DOUBLE PRECISION` for `dsyr2` Specifies the scalar *alpha*.
x	`REAL` for `ssyr2` `DOUBLE PRECISION` for `dsyr2` Array, `DIMENSION` at least `(1 + (`*n*` - 1)*abs(`*incx*`))`. Before entry, the incremented array *x* must contain the *n*-element vector *x*.
incx	`INTEGER`. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.
y	`REAL` for `ssyr2` `DOUBLE PRECISION` for `dsyr2` Array, `DIMENSION` at least `(1 + (`*n*` - 1)*abs(`*incy*`))`. Before entry, the incremented array *y* must contain the *n*-element vector *y*.
incy	`INTEGER`. Specifies the increment for the elements of y. The value of *incy* must not be zero.
a	`REAL` for `ssyr2` `DOUBLE PRECISION` for `dsyr2` Array, `DIMENSION (`*lda, n*`)`. Before entry with *uplo* = `'U'` or `'u'`, the leading *n* by *n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = `'L'` or `'l'`, the leading *n* by *n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.
lda	`INTEGER`. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least `max(1,`*n*`)`.

Output Parameters

a — With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.

?tbmv

Computes a matrix-vector product using a triangular band matrix.

Syntax

```
call stbmv ( uplo, trans, diag, n, k, a, lda, x, incx )
call dtbmv ( uplo, trans, diag, n, k, a, lda, x, incx )
call ctbmv ( uplo, trans, diag, n, k, a, lda, x, incx )
call ztbmv ( uplo, trans, diag, n, k, a, lda, x, incx )
```

Description

The ?tbmv routines perform one of the matrix-vector operations defined as

x := a*x, or x := a'*x, or x := conjg(a')*x,

where:

x is an *n*-element vector

a is an *n* by *n* unit, or non-unit, upper or lower triangular band matrix, with (*k* + 1) diagonals.

Input Parameters

uplo — CHARACTER*1. Specifies whether the matrix is an upper or lower triangular matrix, as follows:

uplo value	**Matrix *a***
U or u	An upper triangular matrix.
L or l	A lower triangular matrix.

trans CHARACTER*1. Specifies the operation to be performed, as follows:

trans value	**Operation to be Performed**
N or n	*x* := a**x*
T or t	*x* := *a*'**x*
C or c	*x* := conjg(*a*')**x*

diag CHARACTER*1. Specifies whether or not a is unit triangular, as follows:

diag value	**Matrix *a***
U or u	Matrix *a* is assumed to be unit triangular.
N or n	Matrix *a* is not assumed to be unit triangular.

n INTEGER. Specifies the order of the matrix a. The value of *n* must be at least zero.

k INTEGER. On entry with *uplo* = 'U' or 'u', *k* specifies the number of super-diagonals of the matrix *a*. On entry with *uplo* = 'L' or 'l', *k* specifies the number of sub-diagonals of the matrix *a*. The value of *k* must satisfy $0 \le k$.

a REAL for stbmv
DOUBLE PRECISION for dtbmv
COMPLEX for ctbmv
DOUBLE COMPLEX for ztbmv

Array, DIMENSION (*lda, n*). Before entry with *uplo* = 'U' or 'u', the leading (*k* + 1) by *n* part of the array *a* must contain the upper triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row (*k* + 1) of the array, the first super-diagonal starting at position 2 in row *k*, and so on. The top left *k* by *k* triangle of the array *a* is not referenced. The following program segment transfers an upper triangular band matrix from conventional full matrix storage to band storage:

```
do 20, j = 1, n
 m = k + 1 - j
do 10, i = max(1, j - k), j
```

```
 a(m + i, j) = matrix(i, j)
10 continue
20 continue
```

Before entry with *uplo* = 'L' or 'l', the leading (*k* + 1) by *n* part of the array a must contain the lower triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right *k* by *k* triangle of the array *a* is not referenced. The following program segment transfers a lower triangular band matrix from conventional full matrix storage to band storage:

```
do 20, j = 1, n
 m = 1 - j
 do 10, i = j, min(n, j + k)
  a(m + i, j) = matrix (i, j)
 10 continue
20 continue
```

Note that when *diag* = 'U' or 'u', the elements of the array *a* corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity.

lda INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least (*k* + 1).

x REAL for stbmv
DOUBLE PRECISION for dtbmv
COMPLEX for ctbmv
DOUBLE COMPLEX for ztbmv

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

Output Parameters

x Overwritten with the transformed vector *x*.

?tbsv

Solves a system of linear equations whose coefficients are in a triangular band matrix.

Syntax

```
call stbsv ( uplo, trans, diag, n, k, a, lda, x, incx )
call dtbsv ( uplo, trans, diag, n, k, a, lda, x, incx )
call ctbsv ( uplo, trans, diag, n, k, a, lda, x, incx )
call ztbsv ( uplo, trans, diag, n, k, a, lda, x, incx )
```

Description

The `?tbsv` routines solve one of the following systems of equations:

`a*x = b`, or `a'*x = b`, or `conjg(a')*x = b`,

where:

b and *x* are *n*-element vectors

a is an *n* by *n* unit, or non-unit, upper or lower triangular band matrix, with `(`*k* `+ 1)` diagonals.

The routine does not test for singularity or near-singularity. Such tests must be performed before calling this routine.

Input Parameters

uplo — `CHARACTER*1`. Specifies whether the matrix is an upper or lower triangular matrix, as follows:

uplo **value**	**Matrix *a***
U or u	An upper triangular matrix.
L or l	A lower triangular matrix.

trans — `CHARACTER*1`. Specifies the operation to be performed, as follows:

trans **value**	**Operation to be Performed**
N or n	`a*x = b`

trans value	Operation to be Performed
T or t	`a'*x = b`
C or c	`conjg(a')*x = b`

diag — `CHARACTER*1`. Specifies whether or not *a* is unit triangular, as follows:

diag value	Matrix *a*
U or u	Matrix *a* is assumed to be unit triangular.
N or n	Matrix *a* is not assumed to be unit triangular.

n — `INTEGER`. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

k — `INTEGER`. On entry with *uplo* = 'U' or 'u', *k* specifies the number of super-diagonals of the matrix *a*. On entry with *uplo* = 'L' or 'l', *k* specifies the number of sub-diagonals of the matrix *a*. The value of *k* must satisfy $0 \le k$.

a — `REAL` for `stbsv`
`DOUBLE PRECISION` for `dtbsv`
`COMPLEX` for `ctbsv`
`DOUBLE COMPLEX` for `ztbsv`

Array, `DIMENSION (lda, n)`. Before entry with *uplo* = 'U' or 'u', the leading (*k* + 1) by *n* part of the array *a* must contain the upper triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row (*k* + 1) of the array, the first super-diagonal starting at position 2 in row *k*, and so on. The top left *k* by *k* triangle of the array *a* is not referenced.

The following program segment transfers an upper triangular band matrix from conventional full matrix storage to band storage:

```
do 20, j = 1, n
 m = k + 1 - j
 do 10, i = max(1, j - k), j
  a(m + i, j) = matrix (i, j)
 10 continue
20 continue
```

Before entry with *uplo* = 'L' or 'l', the leading (*k* + 1) by *n* part of the array *a* must contain the lower triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right *k* by *k* triangle of the array *a* is not referenced.

The following program segment transfers a lower triangular band matrix from conventional full matrix storage to band storage:

```
do 20, j = 1, n
 m = 1 - j
 do 10, i = j, min(n, j + k)
  a(m + i, j) = matrix (i, j)
 10 continue
20 continue
```

When *diag* = 'U' or 'u', the elements of the array a corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity.

lda — INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least (*k* + 1).

x — REAL for stbsv
DOUBLE PRECISION for dtbsv
COMPLEX for ctbsv
DOUBLE COMPLEX for ztbsv

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element right-hand side vector *b*.

incx — INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

Output Parameters

x — Overwritten with the solution vector *x*.

?tpmv

Computes a matrix-vector product using a triangular packed matrix.

Syntax

```
call stpmv ( uplo, trans, diag, n, ap, x, incx )
```

```
call dtpmv ( uplo, trans, diag, n, ap, x, incx )
call ctpmv ( uplo, trans, diag, n, ap, x, incx )
call ztpmv ( uplo, trans, diag, n, ap, x, incx )
```

Description

The `?tpmv` routines perform one of the matrix-vector operations defined as

x := *a***x*, or *x* := *a*'**x*, or *x* := conjg(*a*')**x*,

where:

x is an *n*-element vector

a is an *n* by *n* unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

Input Parameters

uplo — `CHARACTER*1`. Specifies whether the matrix *a* is an upper or lower triangular matrix, as follows:

uplo value	Matrix *a*
U or u	An upper triangular matrix.
L or l	A lower triangular matrix.

trans — `CHARACTER*1`. Specifies the operation to be performed, as follows:

trans value	Operation To Be Performed
N or n	*x* := *a***x*
T or t	*x* := *a*'**x*
C or c	*x* := conjg(*a*')**x*

diag — `CHARACTER*1`. Specifies whether or not *a* is unit triangular, as follows:

diag value	Matrix *a*
U or u	Matrix *a* is assumed to be unit triangular.
N or n	Matrix *a* is not assumed to be unit triangular.

n — `INTEGER`. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

ap

REAL for stpmv
DOUBLE PRECISION for dtpmv
COMPLEX for ctpmv
DOUBLE COMPLEX for ztpmv

Array, DIMENSION at least ((*n**(*n* + 1))/2). Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a*(1,2) and *a*(2,2) respectively, and so on. Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a*(2,1) and *a*(3,1) respectively, and so on. When *diag* = 'U' or 'u', the diagonal elements of *a* are not referenced, but are assumed to be unity.

x

REAL for stpmv
DOUBLE PRECISION for dtpmv
COMPLEX for ctpmv
DOUBLE COMPLEX for ztpmv

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx

INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

Output Parameters

x

Overwritten with the transformed vector *x*.

?tpsv

Solves a system of linear equations whose coefficients are in a triangular packed matrix.

Syntax

```
call stpsv ( uplo, trans, diag, n, ap, x, incx )
call dtpsv ( uplo, trans, diag, n, ap, x, incx )
call ctpsv ( uplo, trans, diag, n, ap, x, incx )
call ztpsv ( uplo, trans, diag, n, ap, x, incx )
```

Description

The ?tpsv routines solve one of the following systems of equations

*a***x* = *b*, or *a*'**x* = b, or conjg(*a*')**x* = *b*,

where:

b and *x* are *n*-element vectors

a is an *n* by *n* unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

This routine does not test for singularity or near-singularity. Such tests must be performed before calling this routine.

Input Parameters

uplo CHARACTER*1. Specifies whether the matrix *a* is an upper or lower triangular matrix, as follows:

uplo value	Matrix *a*
U or u	An upper triangular matrix.
L or l	A lower triangular matrix.

trans CHARACTER*1. Specifies the operation to be performed, as follows:

trans value	Operation To Be Performed
N or n	*a***x* = *b*
T or t	*a*'**x* = *b*
C or c	conjg(*a*')**x* = *b*

diag CHARACTER*1. Specifies whether or not *a* is unit triangular, as follows:

diag value	Matrix *a*
U or u	Matrix *a* is assumed to be unit triangular.
N or n	Matrix *a* is not assumed to be unit triangular.

n INTEGER. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

ap REAL for stpsv
DOUBLE PRECISION for dtpsv
COMPLEX for ctpsv
DOUBLE COMPLEX for ztpsv

Array, DIMENSION at least ((*n**(*n* + 1))/2). Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(1, 2) and *a*(2, 2) respectively, and so on. Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(2, 1) and *a*(3, 1) respectively, and so on. When *diag* = 'U' or 'u', the diagonal elements of *a* are not referenced, but are assumed to be unity.

x REAL for stpsv
DOUBLE PRECISION for dtpsv
COMPLEX for ctpsv
DOUBLE COMPLEX for ztpsv

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element right-hand side vector *b*.

incx INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

Output Parameters

x Overwritten with the solution vector *x*.

?trmv

Computes a matrix-vector product using a triangular matrix.

Syntax

```
call strmv ( uplo, trans, diag, n, a, lda, x, incx )
call dtrmv ( uplo, trans, diag, n, a, lda, x, incx )
call ctrmv ( uplo, trans, diag, n, a, lda, x, incx )
call ztrmv ( uplo, trans, diag, n, a, lda, x, incx )
```

Description

The ?trmv routines perform one of the following matrix-vector operations defined as

x := *a***x* or *x* := *a*'**x* or *x* := conjg(*a*')**x*,

where:

x is an *n*-element vector

a is an *n* by *n* unit, or non-unit, upper or lower triangular matrix.

Input Parameters

uplo CHARACTER*1. Specifies whether the matrix *a* is an upper or lower triangular matrix, as follows:

uplo value	Matrix *a*
U or u	An upper triangular matrix.
L or l	A lower triangular matrix.

trans CHARACTER*1. Specifies the operation to be performed, as follows:

trans value	Operation To Be Performed
N or n	*x* := *a***x*
T or t	*x* := *a*'**x*
C or c	*x* := conjg(*a*')**x*

diag CHARACTER*1. Specifies whether or not *a* is unit triangular, as follows:

diag value	Matrix *a*
U or u	Matrix *a* is assumed to be unit triangular.
N or n	Matrix *a* is not assumed to be unit triangular.

n INTEGER. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

a REAL for strmv
DOUBLE PRECISION for dtrmv
COMPLEX for ctrmv
DOUBLE COMPLEX for ztrmv

	Array, `DIMENSION (lda,n)`. Before entry with *uplo* = 'U' or 'u', the leading *n* by *n* upper triangular part of the array *a* must contain the upper triangular matrix and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *n* by *n* lower triangular part of the array *a* must contain the lower triangular matrix and the strictly upper triangular part of *a* is not referenced. When *diag* = 'U' or 'u', the diagonal elements of *a* are not referenced either, but are assumed to be unity.
lda	`INTEGER`. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least `max(1, n)`.
x	`REAL` for `strmv` `DOUBLE PRECISION` for `dtrmv` `COMPLEX` for `ctrmv` `DOUBLE COMPLEX` for `ztrmv` Array, `DIMENSION` at least `(1 + (n - 1)*abs(incx))`. Before entry, the incremented array *x* must contain the *n*-element vector *x*.
incx	`INTEGER`. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

Output Parameters

x	Overwritten with the transformed vector *x*.

?trsv

Solves a system of linear equations whose coefficients are in a triangular matrix.

Syntax

```
call strsv ( uplo, trans, diag, n, a, lda, x, incx )
call dtrsv ( uplo, trans, diag, n, a, lda, x, incx )
call ctrsv ( uplo, trans, diag, n, a, lda, x, incx )
call ztrsv ( uplo, trans, diag, n, a, lda, x, incx )
```

Description

The?`trsv` routines solve one of the systems of equations:

*a***x* = *b* or *a*'**x* = *b*, or conjg(*a*')**x* = *b*,

where:

b and *x* are *n*-element vectors

a is an *n* by *n* unit, or non-unit, upper or lower triangular matrix.

The routine does not test for singularity or near-singularity. Such tests must be performed before calling this routine.

Input Parameters

uplo — CHARACTER*1. Specifies whether the matrix is an upper or lower triangular matrix, as follows:

uplo value	Matrix *a*
U or u	An upper triangular matrix.
L or l	A lower triangular matrix.

trans — CHARACTER*1. Specifies the operation to be performed, as follows:

trans value	Operation To Be Performed
N or n	*a***x* = *b*
T or t	*a*'**x* = *b*
C or c	conjg(*a*')**x* = *b*

diag — CHARACTER*1. Specifies whether or not *a* is unit triangular, as follows:

diag value	Matrix *a*
U or u	Matrix *a* is assumed to be unit triangular.
N or n	Matrix *a* is not assumed to be unit triangular.

n — INTEGER. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

a REAL for strsv
DOUBLE PRECISION for dtrsv
COMPLEX for ctrsv
DOUBLE COMPLEX for ztrsv

Array, DIMENSION (*lda*,*n*). Before entry with *uplo* = 'U' or 'u', the leading *n* by *n* upper triangular part of the array *a* must contain the upper triangular matrix and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *n* by *n* lower triangular part of the array *a* must contain the lower triangular matrix and the strictly upper triangular part of *a* is not referenced. When *diag* = 'U' or 'u', the diagonal elements of *a* are not referenced either, but are assumed to be unity.

lda INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *n*).

x REAL for strsv
DOUBLE PRECISION for dtrsv
COMPLEX for ctrsv
DOUBLE COMPLEX for ztrsv

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element right-hand side vector *b*.

incx INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

Output Parameters

x Overwritten with the solution vector *x*.

BLAS Level 3 Routines

BLAS Level 3 routines perform matrix-matrix operations. Table 2-3 lists the BLAS Level 3 routine groups and the data types associated with them.

Table 2-3 BLAS Level 3 Routine Groups and Their Data Types

Routine Group	Data Types	Description
?gemm	s, d, c, z	Matrix-matrix product of general matrices
?hemm	c, z	Matrix-matrix product of Hermitian matrices
?herk	c, z	Rank-k update of Hermitian matrices
?her2k	c, z	Rank-2k update of Hermitian matrices
?symm	s, d, c, z	Matrix-matrix product of symmetric matrices
?syrk	s, d, c, z	Rank-k update of symmetric matrices
?syr2k	s, d, c, z	Rank-2k update of symmetric matrices
?trmm	s, d, c, z	Matrix-matrix product of triangular matrices
?trsm	s, d, c, z	Linear matrix-matrix solution for triangular matrices

Symmetric Multiprocessing Version of Intel® MKL

Many applications spend considerable time for executing BLAS level 3 routines. This time can be scaled by the number of processors available on the system through using the symmetric multiprocessing (SMP) feature built into the Intel MKL Library. The performance enhancements based on the parallel use of the processors are available without any programming effort on your part.

To enhance performance, the library uses the following methods:

- The operation of BLAS level 3 matrix-matrix functions permits to restructure the code in a way which increases the localization of data reference, enhances cache memory use, and reduces the dependency on the memory bus.
- Once the code has been effectively blocked as described above, one of the matrices is distributed across the processors to be multiplied by the second matrix. Such distribution ensures effective cache management which reduces the dependency on the memory bus performance and brings good scaling results.

?gemm

Computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product.

Syntax

```
call sgemm (transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call dgemm (transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call cgemm (transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call zgemm (transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
```

Description

The `?gemm` routines perform a matrix-matrix operation with general matrices. The operation is defined as

`c := alpha*op(a)*op(b) + beta*c,`

where:

`op(x)` is one of `op(x) = x` or `op(x) = x'` or `op(x) = conjg(x')`,

alpha and *beta* are scalars

a, *b* and *c* are matrices:

`op(a)` is an *m* by *k* matrix

`op(b)` is a *k* by *n* matrix

c is an *m* by *n* matrix.

Input Parameters

transa `CHARACTER*1`. Specifies the form of `op(a)` to be used in the matrix multiplication as follows:

transa value	Form of `op(a)`
N or n	`op(a) = a`
T or t	`op(a) = a'`
C or c	`op(a) = conjg(a')`

transsb

`CHARACTER*1`. Specifies the form of `op(b)` to be used in the matrix multiplication as follows:

transb **value**	**Form of** `op(b)`
`N` or `n`	`op(b) = b`
`T` or `t`	`op(b) = b'`
`C` or `c`	`op(b) = conjg(b')`

m

`INTEGER`. Specifies the number of rows of the matrix `op(a)` and of the matrix *c*. The value of *m* must be at least zero.

n

`INTEGER`. Specifies the number of columns of the matrix `op(b)` and the number of columns of the matrix *c*. The value of *n* must be at least zero.

k

`INTEGER`. Specifies the number of columns of the matrix `op(a)` and the number of rows of the matrix `op(b)`. The value of *k* must be at least zero.

alpha

`REAL` for `sgemm`
`DOUBLE PRECISION` for `dgemm`
`COMPLEX` for `cgemm`
`DOUBLE COMPLEX` for `zgemm`

Specifies the scalar *alpha*.

a

`REAL` for `sgemm`
`DOUBLE PRECISION` for `dgemm`
`COMPLEX` for `cgemm`
`DOUBLE COMPLEX` for `zgemm`

Array, `DIMENSION (`*lda, ka*`)`, where *ka* is *k* when *transa* = `'N'` or `'n'`, and is *m* otherwise. Before entry with *transa* = `'N'` or `'n'`, the leading *m* by *k* part of the array *a* must contain the matrix *a*, otherwise the leading *k* by *m* part of the array *a* must contain the matrix *a*.

lda

`INTEGER`. Specifies the first dimension of *a* as declared in the calling (sub)program. When *transa* = `'N'` or `'n'`, then *lda* must be at least `max(1, `*m*`)`, otherwise *lda* must be at least `max(1, `*k*`)`.

b

`REAL` for `sgemm`
`DOUBLE PRECISION` for `dgemm`
`COMPLEX` for `cgemm`
`DOUBLE COMPLEX` for `zgemm`

Array, DIMENSION (*ldb, kb*), where *kb* is *n* when *transb* = 'N' or 'n', and is *k* otherwise. Before entry with *transb* = 'N' or 'n', the leading *k* by *n* part of the array *b* must contain the matrix *b*, otherwise the leading *n* by *k* part of the array *b* must contain the matrix *b*.

ldb INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program. When *transb* = 'N' or 'n', then *ldb* must be at least max(1, *k*), otherwise *ldb* must be at least max(1, *n*).

beta REAL for sgemm
DOUBLE PRECISION for dgemm
COMPLEX for cgemm
DOUBLE COMPLEX for zgemm

Specifies the scalar *beta*. When *beta* is supplied as zero, then *c* need not be set on input.

c REAL for sgemm
DOUBLE PRECISION for dgemm
COMPLEX for cgemm
DOUBLE COMPLEX for zgemm

Array, DIMENSION (*ldc, n*). Before entry, the leading *m* by *n* part of the array *c* must contain the matrix *c*, except when *beta* is zero, in which case *c* need not be set on entry.

ldc INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least max(1, *m*).

Output Parameters

c Overwritten by the *m* by *n* matrix (*alpha**op(*a*)*op(*b*) + *beta***c*).

?hemm

Computes a scalar-matrix-matrix product (either one of the matrices is Hermitian) and adds the result to scalar-matrix product.

Syntax

```
call chemm ( side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc )
```

```
call zhemm ( side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc )
```

Description

The `?hemm` routines perform a matrix-matrix operation using Hermitian matrices. The operation is defined as

*c := alpha*a*b + beta*c*

or

*c := alpha*b*a + beta*c,*

where:

alpha and *beta* are scalars

a is an Hermitian matrix

b and *c* are *m* by *n* matrices.

Input Parameters

side — `CHARACTER*1`. Specifies whether the Hermitian matrix *a* appears on the left or right in the operation as follows:

side value	Operation To Be Performed
L or l	*c := alpha*a*b + beta*c*
R or r	*c := alpha*b*a + beta*c*

uplo — `CHARACTER*1`. Specifies whether the upper or lower triangular part of the Hermitian matrix *a* is to be referenced as follows:

uplo value	Part of Matrix *a* To Be Referenced
U or u	Only the upper triangular part of the Hermitian matrix is to be referenced.
L or l	Only the lower triangular part of the Hermitian matrix is to be referenced.

m — `INTEGER`. Specifies the number of rows of the matrix *c*. The value of *m* must be at least zero.

n — `INTEGER`. Specifies the number of columns of the matrix *c*. The value of *n* must be at least zero.

alpha

COMPLEX for chemm
DOUBLE COMPLEX for zhemm

Specifies the scalar *alpha*.

a

COMPLEX for chemm
DOUBLE COMPLEX for zhemm

Array, DIMENSION (*lda*, *ka*), where *ka* is *m* when *side* = 'L' or 'l' and is *n* otherwise. Before entry with *side* = 'L' or 'l', the *m* by *m* part of the array *a* must contain the Hermitian matrix, such that when *uplo* = 'U' or 'u', the leading *m* by *m* upper triangular part of the array *a* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *a* is not referenced, and when *uplo* = 'L' or 'l', the leading *m* by *m* lower triangular part of the array *a* must contain the lower triangular part of the Hermitian matrix, and the strictly upper triangular part of *a* is not referenced. Before entry with *side* = 'R' or 'r', the *n* by *n* part of the array *a* must contain the Hermitian matrix, such that when *uplo* = 'U' or 'u', the leading *n* by *n* upper triangular part of the array *a* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *a* is not referenced, and when *uplo* = 'L' or 'l', the leading *n* by *n* lower triangular part of the array *a* must contain the lower triangular part of the Hermitian matrix, and the strictly upper triangular part of *a* is not referenced. The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.

lda

INTEGER. Specifies the first dimension of *a* as declared in the calling (sub) program. When *side* = 'L' or 'l' then *lda* must be at least max(1, *m*), otherwise *lda* must be at least max(1,*n*).

b

COMPLEX for chemm
DOUBLE COMPLEX for zhemm

Array, DIMENSION (*ldb*, *n*). Before entry, the leading *m* by *n* part of the array *b* must contain the matrix *b*.

ldb

INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program. The value of *ldb* must be at least max(1, *m*).

beta COMPLEX for chemm
DOUBLE COMPLEX for zhemm

Specifies the scalar *beta*. When *beta* is supplied as zero, then *c* need not be set on input.

c COMPLEX for chemm
DOUBLE COMPLEX for zhemm

Array, DIMENSION (*c*, *n*). Before entry, the leading *m* by *n* part of the array *c* must contain the matrix c, except when *beta* is zero, in which case *c* need not be set on entry.

ldc INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least max(1,*m*).

Output Parameters

c Overwritten by the *m* by *n* updated matrix.

?herk

Performs a rank-n update of a Hermitian matrix.

Syntax

```
call cherk ( uplo, trans, n, k, alpha, a, lda, beta, c, ldc )
call zherk ( uplo, trans, n, k, alpha, a, lda, beta, c, ldc )
```

Description

The ?herk routines perform a matrix-matrix operation using Hermitian matrices. The operation is defined as

c := *alpha***a**conjg(*a*') + *beta***c*,

or

c := *alpha**conjg(*a*')**a* + *beta***c*,

where:

alpha and *beta* are real scalars

c is an *n* by *n* Hermitian matrix

a is an *n* by *k* matrix in the first case and a *k* by *n* matrix in the second case.

Input Parameters

uplo — CHARACTER*1. Specifies whether the upper or lower triangular part of the array *c* is to be referenced as follows:

uplo value	Part of Array *c* To Be Referenced
U or u	Only the upper triangular part of *C* is to be referenced.
L or l	Only the lower triangular part of *C* is to be referenced.

trans — CHARACTER*1. Specifies the operation to be performed as follows:

trans value	Operation to be Performed
N or n	*c*:= *alpha***a**conjg(*a*')+*beta***c*
C or c	*c*:= *alpha**conjg(*a*')**a*+*beta***c*

n — INTEGER. Specifies the order of the matrix *c*. The value of *n* must be at least zero.

k — INTEGER. With *trans* = 'N' or 'n', *k* specifies the number of columns of the matrix *a*, and with *trans* = 'C' or 'c', *k* specifies the number of rows of the matrix *a*. The value of *k* must be at least zero.

alpha — REAL for cherk
DOUBLE PRECISION for zherk

Specifies the scalar *alpha*.

a — COMPLEX for cherk
DOUBLE COMPLEX for zherk

Array, DIMENSION (*lda*, *ka*), where *ka* is *k* when *trans* = 'N' or 'n', and is *n* otherwise. Before entry with *trans* = 'N' or 'n', the leading *n* by *k* part of the array *a* must contain the matrix *a*, otherwise the leading *k* by *n* part of the array *a* must contain the matrix *a*.

lda — INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. When *trans* = 'N' or 'n', then *lda* must be at least max(1, *n*), otherwise *lda* must be at least max(1, *k*).

beta	`REAL` for `cherk` `DOUBLE PRECISION` for `zherk` Specifies the scalar *beta*.
c	`COMPLEX` for `cherk` `DOUBLE COMPLEX` for `zherk` Array, `DIMENSION` (*ldc*,*n*). Before entry with *uplo* = `'U'` or `'u'`, the leading *n* by *n* upper triangular part of the array *c* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *c* is not referenced. Before entry with *uplo* = `'L'` or `'l'`, the leading *n* by *n* lower triangular part of the array *c* must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of *c* is not referenced. The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.
ldc	`INTEGER`. Specifies the first dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least `max(1,` *n*`)`.

Output Parameters

c	With *uplo* = `'U'` or `'u'`, the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix. With *uplo* = `'L'` or `'l'`, the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix. The imaginary parts of the diagonal elements are set to zero.

?her2k

Performs a rank-2k update of a Hermitian matrix.

Syntax

```
call cher2k ( uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc )
call zher2k ( uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc )
```

Description

The ?her2k routines perform a rank-2k matrix-matrix operation using Hermitian matrices. The operation is defined as

c := alpha*a*conjg(b') + conjg(alpha)*b*conjg(a') + beta*c,

or

c := alpha*conjg(b')*a + conjg(alpha)*conjg(a')*b + beta*c,

where:

alpha is a scalar and *beta* is a real scalar

c is an *n* by *n* Hermitian matrix

a and *b* are *n* by *k* matrices in the first case and *k* by *n* matrices in the second case.

Input Parameters

uplo CHARACTER*1. Specifies whether the upper or lower triangular part of the array *c* is to be referenced as follows:

uplo value	Part of Array *c* To Be Referenced
U or u	Only the upper triangular part of *C* is to be referenced.
L or l	Only the lower triangular part of *C* is to be referenced.

trans CHARACTER*1. Specifies the operation to be performed as follows:

trans value	Operation to be Performed
N or n	c:=alpha*a*conjg(b') +alpha*b*conjg(a') +beta*c
C or c	c:=alpha*conjg(a')*b +alpha*conjg(b')*a+beta*c

n INTEGER. Specifies the order of the matrix *c*. The value of *n* must be at least zero.

k INTEGER. With *trans* = 'N' or 'n', *k* specifies the number of columns of the matrix *a*, and with *trans* = 'C' or 'c', *k* specifies the number of rows of the matrix *a*. The value of *k* must be at least zero.

alpha — COMPLEX for cher2k
DOUBLE COMPLEX for zher2k

Specifies the scalar *alpha*.

a — COMPLEX for cher2k
DOUBLE COMPLEX for zher2k

Array, DIMENSION (*lda, ka*), where *ka* is *k* when *trans* = 'N' or 'n', and is *n* otherwise. Before entry with *trans* = 'N' or 'n', the leading *n* by *k* part of the array *a* must contain the matrix *a*, otherwise the leading *k* by *n* part of the array *a* must contain the matrix *a*.

lda — INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. When *trans* = 'N' or 'n', then *lda* must be at least max(1, *n*), otherwise *lda* must be at least max(1, *k*).

beta — REAL for cher2k
DOUBLE PRECISION for zher2k

Specifies the scalar *beta*.

b — COMPLEX for cher2k
DOUBLE COMPLEX for zher2k

Array, DIMENSION (*ldb, kb*), where *kb* is *k* when *trans* = 'N' or 'n', and is *n* otherwise. Before entry with *trans* = 'N' or 'n', the leading *n* by *k* part of the array *b* must contain the matrix *b*, otherwise the leading *k* by *n* part of the array *b* must contain the matrix *b*.

ldb — INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program. When *trans* = 'N' or 'n', then *ldb* must be at least max(1, *n*), otherwise *ldb* must be at least max(1, *k*).

c — COMPLEX for cher2k
DOUBLE COMPLEX for zher2k

Array, DIMENSION (*ldc,n*). Before entry with *uplo* = 'U' or 'u', the leading *n* by *n* upper triangular part of the array *c* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *c* is not referenced.

Before entry with *uplo* = 'L' or 'l', the leading *n* by *n* lower triangular part of the array *c* must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of *c* is not referenced.

The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.

ldc — `INTEGER`. Specifies the first dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least `max(1, n)`.

Output Parameters

c — With *uplo* = `'U'` or `'u'`, the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = `'L'` or `'l'`, the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements are set to zero.

?symm

Performs a scalar-matrix-matrix product (one matrix operand is symmetric) and adds the result to a scalar-matrix product.

Syntax

```
call ssymm ( side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc )
call dsymm ( side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc )
call csymm ( side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc )
call zsymm ( side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc )
```

Description

The `?symm` routines perform a matrix-matrix operation using symmetric matrices. The operation is defined as

*c := alpha*a*b + beta*c,*

or

*c := alpha*b*a + beta*c,*

where:

alpha and *beta* are scalars

a is a symmetric matrix

b and *c* are *m* by *n* matrices.

Input Parameters

side CHARACTER*1. Specifies whether the symmetric matrix *a* appears on the left or right in the operation as follows:

side value	Operation to be Performed
L or l	*c* := *alpha***a***b* + *beta***c*
R or r	*c* := *alpha***b***a* + *beta***c*

uplo CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix *a* is to be referenced as follows:

uplo value	Part of Array *a* To Be Referenced
U or u	Only the upper triangular part of the symmetric matrix is to be referenced.
L or l	Only the lower triangular part of the symmetric matrix is to be referenced.

m INTEGER. Specifies the number of rows of the matrix *c*. The value of *m* must be at least zero.

n INTEGER. Specifies the number of columns of the matrix *c*. The value of *n* must be at least zero.

alpha REAL for ssymm
DOUBLE PRECISION for dsymm
COMPLEX for csymm
DOUBLE COMPLEX for zsymm

Specifies the scalar *alpha*.

a REAL for ssymm
DOUBLE PRECISION for dsymm
COMPLEX for csymm
DOUBLE COMPLEX for zsymm

Array, DIMENSION (*lda*, *ka*), where *ka* is *m* when *side* = 'L' or 'l' and is n otherwise. Before entry with *side* = 'L' or 'l', the *m* by *m* part of the array *a* must contain the symmetric matrix, such that when *uplo* = 'U' or 'u', the leading *m* by *m* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced, and when *uplo* = 'L' or 'l', the leading *m* by *m* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.

Before entry with *side* = 'R' or 'r', the *n* by *n* part of the array *a* must contain the symmetric matrix, such that when *uplo* = 'U' or 'u', the leading *n* by *n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced, and when *uplo* = 'L' or 'l', the leading *n* by *n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.

lda — INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. When *side* = 'L' or 'l' then *lda* must be at least max(1, *m*), otherwise *lda* must be at least max(1, *n*).

b — REAL for ssymm
DOUBLE PRECISION for dsymm
COMPLEX for csymm
DOUBLE COMPLEX for zsymm

Array, DIMENSION (*ldb*,*n*). Before entry, the leading *m* by *n* part of the array b must contain the matrix *b*.

ldb — INTEGER. Specifies the first dimension of b as declared in the calling (sub)program. The value of *ldb* must be at least max(1, *m*).

beta — REAL for ssymm
DOUBLE PRECISION for dsymm
COMPLEX for csymm
DOUBLE COMPLEX for zsymm

Specifies the scalar *beta*. When *beta* is supplied as zero, then *c* need not be set on input.

c — REAL for ssymm
DOUBLE PRECISION for dsymm
COMPLEX for csymm
DOUBLE COMPLEX for zsymm

Array, DIMENSION (*ldc*,*n*). Before entry, the leading *m* by *n* part of the array *c* must contain the matrix *c*, except when beta is zero, in which case *c* need not be set on entry.

ldc INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least max(1, *m*).

Output Parameters

c Overwritten by the *m* by *n* updated matrix.

?syrk

Performs a rank-n update of a symmetric matrix.

Syntax

```
call ssyrk ( uplo, trans, n, k, alpha, a, lda, beta, c, ldc )
call dsyrk ( uplo, trans, n, k, alpha, a, lda, beta, c, ldc )
call csyrk ( uplo, trans, n, k, alpha, a, lda, beta, c, ldc )
call zsyrk ( uplo, trans, n, k, alpha, a, lda, beta, c, ldc )
```

Description

The ?syrk routines perform a matrix-matrix operation using symmetric matrices. The operation is defined as

c := *alpha***a***a*' + *beta***c*,

or

c := *alpha***a*'**a* + *beta***c*,

where:

alpha and *beta* are scalars

c is an *n* by *n* symmetric matrix

a is an *n* by *k* matrix in the first case and a *k* by *n* matrix in the second case.

Input Parameters

uplo — CHARACTER*1. Specifies whether the upper or lower triangular part of the array *c* is to be referenced as follows:

uplo value	Part of Array *c* To Be Referenced
U or u	Only the upper triangular part of *c* is to be referenced.
L or l	Only the lower triangular part of *c* is to be referenced.

trans — CHARACTER*1. Specifies the operation to be performed as follows:

trans value	Operation to be Performed
N or n	*c:= alpha*a*a' + beta*c*
T or t	*c:= alpha*a'*a + beta*c*
C or c	*c:= alpha*a'*a + beta*c*

n — INTEGER. Specifies the order of the matrix *c*. The value of *n* must be at least zero.

k — INTEGER. On entry with *trans* = 'N' or 'n', *k* specifies the number of columns of the matrix *a*, and on entry with *trans* = 'T' or 't' or 'C' or 'c', *k* specifies the number of rows of the matrix *a*. The value of *k* must be at least zero.

alpha — REAL for ssyrk
DOUBLE PRECISION for dsyrk
COMPLEX for csyrk
DOUBLE COMPLEX for zsyrk

Specifies the scalar *alpha*.

a — `REAL` for `ssyrk`
`DOUBLE PRECISION` for `dsyrk`
`COMPLEX` for `csyrk`
`DOUBLE COMPLEX` for `zsyrk`

Array, `DIMENSION` (*lda*, *ka*), where *ka* is *k* when *trans* = `'N'` or `'n'`, and is *n* otherwise. Before entry with *trans* = `'N'` or `'n'`, the leading *n* by *k* part of the array *a* must contain the matrix *a*, otherwise the leading *k* by *n* part of the array *a* must contain the matrix *a*.

lda — `INTEGER`. Specifies the first dimension of *a* as declared in the calling (sub)program. When *trans* = `'N'` or `'n'`, then *lda* must be at least `max(1,`*n*`)`, otherwise *lda* must be at least `max(1, `*k*`)`.

beta — `REAL` for `ssyrk`
`DOUBLE PRECISION` for `dsyrk`
`COMPLEX` for `csyrk`
`DOUBLE COMPLEX` for `zsyrk`

Specifies the scalar *beta*.

c — `REAL` for `ssyrk`
`DOUBLE PRECISION` for `dsyrk`
`COMPLEX` for `csyrk`
`DOUBLE COMPLEX` for `zsyrk`

Array, `DIMENSION` (*ldc*, *n*). Before entry with *uplo* = `'U'` or `'u'`, the leading *n* by *n* upper triangular part of the array `c` must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *c* is not referenced.

Before entry with *uplo* = `'L'` or `'l'`, the leading *n* by *n* lower triangular part of the array *c* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *c* is not referenced.

ldc — `INTEGER`. Specifies the first dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least `max(1, `*n*`)`.

Output Parameters

c — With *uplo* = `'U'` or `'u'`, the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = `'L'` or `'l'`, the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix.

?syr2k

Performs a rank-2k update of a symmetric matrix.

Syntax

```
call ssyr2k ( uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc )
call dsyr2k ( uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc )
call csyr2k ( uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc )
call zsyr2k ( uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc )
```

Description

The `?syr2k` routines perform a rank-2k matrix-matrix operation using symmetric matrices. The operation is defined as

*c := alpha*a*b' + alpha*b*a' + beta*c,*

or

*c := alpha*a'*b + alpha*b'*a + beta*c,*

where:

alpha and *beta* are scalars

c is an *n* by *n* symmetric matrix

a and *b* are *n* by *k* matrices in the first case and *k* by *n* matrices in the second case.

Input Parameters

uplo `CHARACTER*1`. Specifies whether the upper or lower triangular part of the array *c* is to be referenced as follows:

uplo value	Part of Array *c* To Be Referenced
U or u	Only the upper triangular part of *c* is to be referenced.
L or l	Only the lower triangular part of *c* is to be referenced.

trans — `CHARACTER*1`. Specifies the operation to be performed as follows:

`trans` **value**	**Operation to be Performed**
`N` or `n`	*c*:= *alpha*a*b'+alpha*b*a'+beta*c*
`T` or `t`	*c*:= *alpha*a'*b+alpha*b'*a+beta*c*
`C` or `c`	*c*:= *alpha*a'*b+alpha*b'*a+beta*c*

n — `INTEGER`. Specifies the order of the matrix *c*. The value of *n* must be at least zero.

k — `INTEGER`. On entry with *trans* = `'N'` or `'n'`, *k* specifies the number of columns of the matrices *a* and *b*, and on entry with *trans* = `'T'` or `'t'` or `'C'` or `'c'`, *k* specifies the number of rows of the matrices *a* and *b*. The value of *k* must be at least zero.

alpha — `REAL` for `ssyr2k`
`DOUBLE PRECISION` for `dsyr2k`
`COMPLEX` for `csyr2k`
`DOUBLE COMPLEX` for `zsyr2k`

Specifies the scalar *alpha*.

a — `REAL` for `ssyr2k`
`DOUBLE PRECISION` for `dsyr2k`
`COMPLEX` for `csyr2k`
`DOUBLE COMPLEX` for `zsyr2k`

Array, `DIMENSION (`*lda, ka*`)`, where *ka* is *k* when *trans* = `'N'` or `'n'`, and is *n* otherwise. Before entry with *trans* = `'N'` or `'n'`, the leading *n* by *k* part of the array *a* must contain the matrix *a*, otherwise the leading *k* by *n* part of the array *a* must contain the matrix *a*.

lda — `INTEGER`. Specifies the first dimension of *a* as declared in the calling (sub)program. When *trans* = `'N'` or `'n'`, then *lda* must be at least `max(1,`*n*`)`, otherwise *lda* must be at least `max(1, `*k*`)`.

b — `REAL` for `ssyr2k`
`DOUBLE PRECISION` for `dsyr2k`
`COMPLEX` for `csyr2k`
`DOUBLE COMPLEX` for `zsyr2k`

Array, DIMENSION *(ldb, kb)* where *kb* is *k* when *trans* = 'N' or 'n' and is 'n' otherwise. Before entry with *trans* = 'N' or 'n', the leading *n* by *k* part of the array *b* must contain the matrix *b*, otherwise the leading *k* by *n* part of the array *b* must contain the matrix *b*.

ldb INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. When *trans* = 'N' or 'n', then *ldb* must be at least max(1,*n*), otherwise *ldb* must be at least max(1, *k*).

beta REAL for ssyr2k
DOUBLE PRECISION for dsyr2k
COMPLEX for csyr2k
DOUBLE COMPLEX for zsyr2k

Specifies the scalar *beta*.

c REAL for ssyr2k
DOUBLE PRECISION for dsyr2k
COMPLEX for csyr2k
DOUBLE COMPLEX for zsyr2k

Array, DIMENSION (*ldc*,*n*). Before entry with *uplo* = 'U' or 'u', the leading *n* by *n* upper triangular part of the array c must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *c* is not referenced.

Before entry with *uplo* = 'L' or 'l', the leading *n* by *n* lower triangular part of the array *c* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *c* is not referenced.

ldc INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least max(1, *n*).

Output Parameters

c With *uplo* = 'U' or 'u', the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix.

?trmm

Computes a scalar-matrix-matrix product (one matrix operand is triangular).

Syntax

```
call strmm ( side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
call dtrmm ( side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
call ctrmm ( side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
call ztrmm ( side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
```

Description

The ?trmm routines perform a matrix-matrix operation using triangular matrices. The operation is defined as

b := *alpha**op(*a*)**b*

or

b := *alpha***b**op(*a*)

where:

alpha is a scalar

b is an *m* by *n* matrix

a is a unit, or non-unit, upper or lower triangular matrix

op(*a*) is one of op(*a*) = *a* or op(*a*) = *a*' or op(*a*) = conjg(*a*').

Input Parameters

side CHARACTER*1. Specifies whether op(*a*) multiplies *b* from the left or right in the operation as follows:

side value	Operation To Be Performed
L or l	*b* := *alpha**op(*a*)**b*
R or r	*b* := *alpha***b**op(*a*)

uplo CHARACTER*1. Specifies whether the matrix *a* is an upper or lower triangular matrix as follows:

uplo **value**	**Matrix** *a*
U or u	Matrix *a* is an upper triangular matrix.
L or l	Matrix *a* is a lower triangular matrix.

transa CHARACTER*1. Specifies the form of op(*a*) to be used in the matrix multiplication as follows:

transa **value**	**Form of** op(*a*)
N or n	op(*a*) = *a*
T or t	op(*a*) = *a*'
C or c	op(*a*) = conjg(*a*')

diag CHARACTER*1. Specifies whether or not *a* is unit triangular as follows:

diag **value**	**Matrix** *a*
U or u	Matrix *a* is assumed to be unit triangular.
N or n	Matrix *a* is not assumed to be unit triangular.

m INTEGER. Specifies the number of rows of *b*. The value of *m* must be at least zero.

n INTEGER. Specifies the number of columns of *b*. The value of *n* must be at least zero.

alpha REAL for strmm
DOUBLE PRECISION for dtrmm
COMPLEX for ctrmm
DOUBLE COMPLEX for ztrmm

Specifies the scalar *alpha*. When *alpha* is zero, then *a* is not referenced and *b* need not be set before entry.

a REAL for strmm
DOUBLE PRECISION for dtrmm
COMPLEX for ctrmm
DOUBLE COMPLEX for ztrmm

Array, `DIMENSION (lda, k)`, where *k* is *m* when *side* = `'L'` or `'l'` and is *n* when *side* = `'R'` or `'r'`. Before entry with *uplo* = `'U'` or `'u'`, the leading *k* by *k* upper triangular part of the array *a* must contain the upper triangular matrix and the strictly lower triangular part of *a* is not referenced.

Before entry with *uplo* = `'L'` or `'l'`, the leading *k* by *k* lower triangular part of the array *a* must contain the lower triangular matrix and the strictly upper triangular part of *a* is not referenced. When *diag* = `'U'` or `'u'`, the diagonal elements of *a* are not referenced either, but are assumed to be unity.

lda — `INTEGER`. Specifies the first dimension of *a* as declared in the calling (sub)program. When *side* = `'L'` or `'l'`, then *lda* must be at least `max(1, m)`, when *side* = `'R'` or `'r'`, then *lda* must be at least `max(1, n)`.

b — `REAL` for `strmm`
`DOUBLE PRECISION` for `dtrmm`
`COMPLEX` for `ctrmm`
`DOUBLE COMPLEX` for `ztrmm`

Array, `DIMENSION (ldb, n)`. Before entry, the leading *m* by *n* part of the array *b* must contain the matrix *b*.

ldb — `INTEGER`. Specifies the first dimension of *b* as declared in the calling (sub)program. The value of *ldb* must be at least `max(1, m)`.

Output Parameters

b — Overwritten by the transformed matrix.

?trsm

Solves a matrix equation (one matrix operand is triangular).

Syntax

```
call strsm ( side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
call dtrsm ( side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
call ctrsm ( side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
call ztrsm ( side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
```

Description

The `?trsm` routines solve one of the following matrix equations:

`op(a)*x = alpha*b,`

or

`x*op(a) = alpha*b,`

where:

alpha is a scalar

x and *b* are *m* by *n* matrices

a is a unit, or non-unit, upper or lower triangular matrix

`op(a)` is one of `op(a) = a` or `op(a) = a'` or
`op(a) = conjg(a')`.

The matrix *x* is overwritten on *b*.

Input Parameters

side — `CHARACTER*1`. Specifies whether `op(a)` appears on the left or right of *x* for the operation to be performed as follows:

side value	Operation To Be Performed
L or l	`op(a)*x = alpha*b`
R or r	`x*op(a) = alpha*b`

uplo — `CHARACTER*1`. Specifies whether the matrix *a* is an upper or lower triangular matrix as follows:

uplo value	Matrix *a*
U or u	Matrix *a* is an upper triangular matrix.
L or l	Matrix *a* is a lower triangular matrix.

transa — `CHARACTER*1`. Specifies the form of `op(a)` to be used in the matrix multiplication as follows:

transa value	Form of `op(a)`
N or n	`op(a) = a`

transa value	Form of op(*a*)
T or t	op(*a*) = *a*'
C or c	op(*a*) = conjg(*a*')

diag CHARACTER*1. Specifies whether or not *a* is unit triangular as follows:

diag value	Matrix *a*
U or u	Matrix *a* is assumed to be unit triangular.
N or n	Matrix *a* is not assumed to be unit triangular.

m INTEGER. Specifies the number of rows of *b*. The value of *m* must be at least zero.

n INTEGER. Specifies the number of columns of *b*. The value of *n* must be at least zero.

alpha REAL for strsm
DOUBLE PRECISION for dtrsm
COMPLEX for ctrsm
DOUBLE COMPLEX for ztrsm

Specifies the scalar *alpha*. When *alpha* is zero, then *a* is not referenced and *b* need not be set before entry.

a REAL for strsm
DOUBLE PRECISION for dtrsm
COMPLEX for ctrsm
DOUBLE COMPLEX for ztrsm

Array, DIMENSION (*lda, k*), where *k* is *m* when *side* = 'L' or 'l' and is *n* when *side* = 'R' or 'r'. Before entry with *uplo* = 'U' or 'u', the leading *k* by *k* upper triangular part of the array *a* must contain the upper triangular matrix and the strictly lower triangular part of *a* is not referenced.

Before entry with *uplo* = 'L' or 'l', the leading *k* by *k* lower triangular part of the array *a* must contain the lower triangular matrix and the strictly upper triangular part of *a* is not referenced. When *diag* = 'U' or 'u', the diagonal elements of *a* are not referenced either, but are assumed to be unity.

lda INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. When *side* = 'L' or 'l', then *lda* must be at least max(1, *m*), when *side* = 'R' or 'r', then *lda* must be at least max(1, *n*).

b REAL for strsm
DOUBLE PRECISION for dtrsm
COMPLEX for ctrsm
DOUBLE COMPLEX for ztrsm

Array, DIMENSION (*ldb*, *n*). Before entry, the leading *m* by *n* part of the array *b* must contain the right-hand side matrix *b*.

ldb INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program. The value of *ldb* must be at least max(1, *m*).

Output Parameters

b Overwritten by the solution matrix *x*.

Sparse BLAS Routines and Functions

This section describes Sparse BLAS, an extension of BLAS Level 1 included in Intel® Math Kernel Library beginning with Intel MKL release 2.1. Sparse BLAS is a group of routines and functions that perform a number of common vector operations on sparse vectors stored in compressed form.

Sparse vectors are those in which the majority of elements are zeros. Sparse BLAS routines and functions are specially implemented to take advantage of vector sparsity. This allows you to achieve large savings in computer time and memory. If *nz* is the number of non-zero vector elements, the computer time taken by Sparse BLAS operations will be $O(nz)$.

Vector Arguments in Sparse BLAS

Compressed sparse vectors. Let *a* be a vector stored in an array, and assume that the only non-zero elements of *a* are the following:

$a(k_1), a(k_2), a(k_3) \ldots a(k_{nz}),$

where *nz* is the total number of non-zero elements in *a*.

In Sparse BLAS, this vector can be represented in compressed form by two FORTRAN arrays, *x* (values) and *indx* (indices). Each array has *nz* elements:

$x(1)=a(k_1), x(2)=a(k_2), \ldots x(nz)=a(k_{nz}),$

$indx(1)=k_1, indx(2)=k_2, \ldots indx(\mathrm{nz})=k_{nz}.$

Thus, a sparse vector is fully determined by the triple (*nz*, *x*, *indx*). If you pass a negative or zero value of *nz* to Sparse BLAS, the subroutines do not modify any arrays or variables.

Full-storage vectors. Sparse BLAS routines can also use a vector argument fully stored in a single FORTRAN array (a full-storage vector). If *y* is a full-storage vector, its elements must be stored contiguously: the first element in *y*(1), the second in *y*(2), and so on. This corresponds to an increment *incy* = 1 in BLAS Level 1. No increment value for full-storage vectors is passed as an argument to Sparse BLAS routines or functions.

Naming Conventions in Sparse BLAS

Similar to BLAS, the names of Sparse BLAS subprograms have prefixes that determine the data type involved: `s` and `d` for single- and double- precision real; `c` and `z` for single- and double-precision complex.

If a Sparse BLAS routine is an extension of a “dense” one, the subprogram name is formed by appending the suffix `i` (standing for *indexed*) to the name of the corresponding “dense” subprogram. For example, the Sparse BLAS routine `saxpyi` corresponds to the BLAS routine `saxpy`, and the Sparse BLAS function `cdotci` corresponds to the BLAS function `cdotc`.

Routines and Data Types in Sparse BLAS

Routines and data types supported in the Intel MKL implementation of Sparse BLAS are listed in Table 2-4.

Table 2-4 Sparse BLAS Routines and Their Data Types

Routine/ Function	Data Types	Description
`?axpyi`	s, d, c, z	Scalar-vector product plus vector (routines)
`?doti`	s, d	Dot product (functions)
`?dotci`	c, z	Complex dot product conjugated (functions)
`?dotui`	c, z	Complex dot product unconjugated (functions)
`?gthr`	s, d, c, z	Gathering a full-storage sparse vector into compressed form: *nz*, *x*, *indx* (routines)
`?gthrz`	s, d, c, z	Gathering a full-storage sparse vector into compressed form and assigning zeros to gathered elements in the full-storage vector (routines)
`?roti`	s, d	Givens rotation (routines)
`?sctr`	s, d, c, z	Scattering a vector from compressed form to full-storage form (routines)

BLAS Routines That Can Work With Sparse Vectors

The following BLAS Level 1 routines will give correct results when you pass to them a compressed-form array `x` (with the increment `incx` = 1):

`?asum`	sum of absolute values of vector elements
`?copy`	copying a vector
`?nrm2`	Euclidean norm of a vector
`?scal`	scaling a vector
`i?amax`	index of the element with the largest absolute value or, for complex flavors, the largest sum $\|\mathrm{Re}\,x(i)\| + \|\mathrm{Im}\,x(i)\|$.
`i?amin`	index of the element with the smallest absolute value or, for complex flavors, the smallest sum $\|\mathrm{Re}\,x(i)\| + \|\mathrm{Im}\,x(i)\|$.

The result *i* returned by `i?amax` and `i?amin` should be interpreted as index in the compressed-form array, so that the largest (smallest) value is `x(i)`; the corresponding index in full-storage array is `indx(i)`.

You can also call `?rotg` to compute the parameters of Givens rotation and then pass these parameters to the Sparse BLAS routines `?roti`.

?axpyi

Adds a scalar multiple of compressed sparse vector to a full-storage vector.

Syntax

```
call saxpyi ( nz, a, x, indx, y )
call daxpyi ( nz, a, x, indx, y )
call caxpyi ( nz, a, x, indx, y )
call zaxpyi ( nz, a, x, indx, y )
```

Description

The `?axpyi` routines perform a vector-vector operation defined as

`y := a*x + y`

where:

a is a scalar

(*nz*, *x*, *indx*) is a sparse vector stored in compressed form

y is a vector in full storage form.

The `?axpyi` routines reference or modify only the elements of *y* whose indices are listed in the array *indx*. The values in *indx* must be distinct.

Input Parameters

nz — `INTEGER`. The number of elements in *x* and *indx* .

a — `REAL` for `saxpyi`
`DOUBLE PRECISION` for `daxpyi`
`COMPLEX` for `caxpyi`
`DOUBLE COMPLEX` for `zaxpyi`

Specifies the scalar *a*.

x — `REAL` for `saxpyi`
`DOUBLE PRECISION` for `daxpyi`
`COMPLEX` for `caxpyi`
`DOUBLE COMPLEX` for `zaxpyi`
Array, `DIMENSION` at least *nz*.

indx — `INTEGER`. Specifies the indices for the elements of *x*.

Array, `DIMENSION` at least *nz*.

y — `REAL` for `saxpyi`
`DOUBLE PRECISION` for `daxpyi`
`COMPLEX` for `caxpyi`
`DOUBLE COMPLEX` for `zaxpyi`

Array, `DIMENSION` at least $\max_i(indx(i))$.

Output Parameters

y — Contains the updated vector *y*.

?doti

Computes the dot product of a compressed sparse real vector by a full-storage real vector.

Syntax

```
res = sdoti ( nz, x, indx, y )
res = ddoti ( nz, x, indx, y )
```

Description

The `?doti` functions return the dot product of *x* and *y* defined as

`x(1)*y(indx(1)) + x(2)*y(indx(2)) +...+ x(nz)*y(indx(nz))`

where the triple (*nz*, *x*, *indx*) defines a sparse real vector stored in compressed form, and *y* is a real vector in full storage form. The functions reference only the elements of *y* whose indices are listed in the array *indx*. The values in *indx* must be distinct.

Input Parameters

nz	`INTEGER`. The number of elements in *x* and *indx*.
x	`REAL` for `sdoti` `DOUBLE PRECISION` for `ddoti` Array, `DIMENSION` at least *nz*.
indx	`INTEGER`. Specifies the indices for the elements of *x*. Array, `DIMENSION` at least *nz*.
y	`REAL` for `sdoti` `DOUBLE PRECISION` for `ddoti` Array, `DIMENSION` at least $\max_i(indx(i))$.

Output Parameters

res	`REAL` for `sdoti` `DOUBLE PRECISION` for `ddoti` Contains the dot product of *x* and *y*, if *nz* is positive. Otherwise, *res* contains `0`.

?dotci

Computes the conjugated dot product of a compressed sparse complex vector with a full-storage complex vector.

Syntax

res = cdotci (*nz*, x, *indx*, *y*)

res = zdotci (*nz*, x, *indx*, *y*)

Description

The ?dotci functions return the dot product of *x* and *y* defined as

conjg(*x*(1))**y*(*indx*(1)) + ... + conjg(*x*(*nz*))**y*(*indx*(*nz*))

where the triple (*nz*, *x*, *indx*) defines a sparse complex vector stored in compressed form, and *y* is a real vector in full storage form. The functions reference only the elements of *y* whose indices are listed in the array *indx*. The values in *indx* must be distinct.

Input Parameters

nz — INTEGER. The number of elements in *x* and *indx* .

x — COMPLEX for cdotci
DOUBLE COMPLEX for zdotci
Array, DIMENSION at least *nz*.

indx — INTEGER. Specifies the indices for the elements of *x*.
Array, DIMENSION at least *nz*.

y — COMPLEX for cdotci
DOUBLE COMPLEX for zdotci
Array, DIMENSION at least $\max_i$(*indx*(*i*)).

Output Parameters

res — COMPLEX for cdotci
DOUBLE COMPLEX for zdotci

Contains the conjugated dot product of *x* and *y*, if *nz* is positive. Otherwise, *res* contains 0.

?dotui

Computes the dot product of a compressed sparse complex vector by a full-storage complex vector.

Syntax

```
res = cdotui ( nz, x, indx, y )
res = zdotui ( nz, x, indx, y )
```

Description

The `?dotui` functions return the dot product of *x* and *y* defined as

`x(1)*y(indx(1)) + x(2)*y(indx(2)) +...+ x(nz)*y(indx(nz))`

where the triple (*nz*, *x*, *indx*) defines a sparse complex vector stored in compressed form, and *y* is a real vector in full storage form. The functions reference only the elements of *y* whose indices are listed in the array *indx*. The values in *indx* must be distinct.

Input Parameters

nz	`INTEGER`. The number of elements in *x* and *indx* .
x	`COMPLEX` for `cdotui` `DOUBLE COMPLEX` for `zdotui` Array, `DIMENSION` at least *nz*.
indx	`INTEGER`. Specifies the indices for the elements of *x*. Array, `DIMENSION` at least *nz*.
y	`COMPLEX` for `cdotui` `DOUBLE COMPLEX` for `zdotui` Array, `DIMENSION` at least $\max_i(indx(i))$.

Output Parameters

res	`COMPLEX` for `cdotui` `DOUBLE COMPLEX` for `zdotui` Contains the dot product of *x* and *y*, if *nz* is positive. Otherwise, *res* contains `0`.

?gthr

Gathers a full-storage sparse vector's elements into compressed form.

Syntax

```
call sgthr ( nz, y, x, indx )
call dgthr ( nz, y, x, indx )
call cgthr ( nz, y, x, indx )
call zgthr ( nz, y, x, indx )
```

Description

The `?gthr` routines gather the specified elements of a full-storage sparse vector *y* into compressed form (*nz*, *x*, *indx*). The routines reference only the elements of *y* whose indices are listed in the array *indx*:

`x(i) = y(indx(i))`, for *i*=1,2,...*nz*.

Input Parameters

nz — `INTEGER`. The number of elements of *y* to be gathered.

indx — `INTEGER`. Specifies indices of elements to be gathered.
Array, `DIMENSION` at least *nz*.

y — `REAL` for `sgthr`
`DOUBLE PRECISION` for `dgthr`
`COMPLEX` for `cgthr`
`DOUBLE COMPLEX` for `zgthr`
Array, `DIMENSION` at least $\max_i$ (*indx*(*i*)).

Output Parameters

x — `REAL` for `sgthr`
`DOUBLE PRECISION` for `dgthr`
`COMPLEX` for `cgthr`
`DOUBLE COMPLEX` for `zgthr`
Array, `DIMENSION` at least *nz*.

Contains the vector converted to the compressed form.

?gthrz

Gathers a sparse vector's elements into compressed form, replacing them by zeros.

Syntax

```
call sgthrz ( nz, y, x, indx )
call dgthrz ( nz, y, x, indx )
call cgthrz ( nz, y, x, indx )
call zgthrz ( nz, y, x, indx )
```

Description

The `?gthrz` routines gather the elements with indices specified by the array *indx* from a full-storage vector *y* into compressed form
(*nz*, *x*, *indx*) and overwrite the gathered elements of *y* by zeros.
Other elements of *y* are not referenced or modified (see also `?gthr`).

Input Parameters

nz	`INTEGER`. The number of elements of *y* to be gathered.
indx	`INTEGER`. Specifies indices of elements to be gathered.Array, `DIMENSION` at least *nz*.
y	`REAL` for `sgthrz` `DOUBLE PRECISION` for `dgthrz` `COMPLEX` for `cgthrz` `DOUBLE COMPLEX` for `zgthrz` Array, `DIMENSION` at least $\max_i$ (*indx*(*i*)).

Output Parameters

x	`REAL` for `sgthrz` `DOUBLE PRECISION` for `dgthrz` `COMPLEX` for `cgthrz` `DOUBLE COMPLEX` for `zgthrz` Array, `DIMENSION` at least *nz*. Contains the vector converted to the compressed form.
y	The updated vector *y*.

?roti

Applies Givens rotation to sparse vectors one of which is in compressed form.

Syntax

```
call sroti ( nz, x, indx, y, c, s )
call droti ( nz, x, indx, y, c, s )
```

Description

The `?roti` routines apply the Givens rotation to elements of two real vectors, *x* (in compressed form *nz*, *x*, *indx*) and *y* (in full storage form):

```
x(i) = c*x(i) + s*y(indx(i))
y(indx(i)) = c*y(indx(i)) - s*x(i)
```

The routines reference only the elements of *y* whose indices are listed in the array *indx*. The values in *indx* must be distinct.

Input Parameters

nz	`INTEGER`. The number of elements in *x* and *indx*.
x	`REAL` for `sroti` `DOUBLE PRECISION` for `droti` Array, `DIMENSION` at least *nz*.
indx	`INTEGER`. Specifies the indices for the elements of *x*. Array, `DIMENSION` at least *nz*.
y	`REAL` for `sroti` `DOUBLE PRECISION` for `droti` Array, `DIMENSION` at least $\max_i$(*indx*(*i*)).
c	A scalar: `REAL` for `sroti` `DOUBLE PRECISION` for `droti`.
s	A scalar: `REAL` for `sroti` `DOUBLE PRECISION` for `droti`.

Output Parameters

x and *y*	The updated arrays.

?sctr

Converts compressed sparse vectors into full storage form.

Syntax

```
call ssctr ( nz, x, indx, y )
call dsctr ( nz, x, indx, y )
call csctr ( nz, x, indx, y )
call zsctr ( nz, x, indx, y )
```

Description

The `?sctr` routines scatter the elements of the compressed sparse vector (*nz*, *x*, *indx*) to a full-storage vector *y*. The routines modify only the elements of *y* whose indices are listed in the array *indx*:

y(*indx*(*i*)) = x(*i*), for *i*=1,2,...*nz*.

Input Parameters

nz — `INTEGER`. The number of elements of *x* to be scattered.

indx — `INTEGER`. Specifies indices of elements to be scattered.Array, `DIMENSION` at least *nz*.

x — `REAL` for `ssctr`
`DOUBLE PRECISION` for `dsctr`
`COMPLEX` for `csctr`
`DOUBLE COMPLEX` for `zsctr`
Array, `DIMENSION` at least *nz*.
Contains the vector to be converted to full-storage form.

Output Parameters

y — `REAL` for `ssctr`
`DOUBLE PRECISION` for `dsctr`
`COMPLEX` for `csctr`
`DOUBLE COMPLEX` for `zsctr`
Array, `DIMENSION` at least $\max_i$ (*indx*(*i*)).
Contains the vector *y* with updated elements.

LAPACK Routines: Linear Equations 3

This chapter describes the Intel® Math Kernel Library implementation of routines from the LAPACK package that are used for solving systems of linear equations and performing a number of related computational tasks. The library includes LAPACK routines for both real and complex data.

Routines are supported for systems of equations with the following types of matrices:

- general
- banded
- symmetric or Hermitian positive-definite (both full and packed storage)
- symmetric or Hermitian positive-definite banded
- symmetric or Hermitian indefinite (both full and packed storage)
- symmetric or Hermitian indefinite banded
- triangular (both full and packed storage)
- triangular banded
- tridiagonal.

For each of the above matrix types, the library includes routines for performing the following computations: *factoring* the matrix (except for triangular matrices); *equilibrating* the matrix; *solving* a system of linear equations; *estimating the condition number* of a matrix; *refining* the solution of linear equations and computing its error bounds; *inverting* the matrix.

To solve a particular problem, you can either call two or more computational routines or call a corresponding driver routine that combines several tasks in one call, such as `?gesv` for factoring and solving. Thus, to solve a system of linear equations with a general matrix, you can first call `?getrf` (*LU* factorization) and then `?getrs` (computing the solution). Then, you might wish to call `?gerfs` to refine the solution and get the error bounds. Alternatively, you can just use the driver routine `?gesvx` which performs all these tasks in one call.

WARNING. LAPACK routines expect that input matrices do not contain `INF` or `NaN` values. When input data is inappropriate for LAPACK, problems may arise, including possible hangs.

Routine Naming Conventions

For each routine introduced in this chapter, you can use the LAPACK name.

LAPACK names are listed in Table 3-1 and Table 3-2, and have the structure `xyyzzz` or `xyyzz`, which is described below.

The initial letter `x` indicates the data type:

`s`	real, single precision	`c`	complex, single precision
`d`	real, double precision	`z`	complex, double precision

The second and third letters `yy` indicate the matrix type and storage scheme:

`ge`	general
`gb`	general band
`gt`	general tridiagonal
`po`	symmetric or Hermitian positive-definite
`pp`	symmetric or Hermitian positive-definite (packed storage)
`pb`	symmetric or Hermitian positive-definite band
`pt`	symmetric or Hermitian positive-definite tridiagonal
`sy`	symmetric indefinite
`sp`	symmetric indefinite (packed storage)
`he`	Hermitian indefinite
`hp`	Hermitian indefinite (packed storage)
`tr`	triangular
`tp`	triangular (packed storage)
`tb`	triangular band

For computational routines, the last three letters `zzz` indicate the computation performed:

`trf`	form a triangular matrix factorization
`trs`	solve the linear system with a factored matrix
`con`	estimate the matrix condition number
`rfs`	refine the solution and compute error bounds
`tri`	compute the inverse matrix using the factorization
`equ`	equilibrate a matrix.

For example, the routine `sgetrf` performs the triangular factorization of general real matrices in single precision; the corresponding routine for complex matrices is `cgetrf`.

For driver routines, the names can end either with `-sv` (meaning a *simple* driver), or with `-svx` (meaning an *expert* driver).

Matrix Storage Schemes

LAPACK routines use the following matrix storage schemes:

- *Full storage*: a matrix A is stored in a two-dimensional array `a`, with the matrix element a_{ij} stored in the array element `a(i,j)`.
- *Packed storage* scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly: the upper or lower triangle of the matrix is packed by columns in a one-dimensional array.
- *Band storage*: an *m* by *n* band matrix with *kl* sub-diagonals and *ku* super-diagonals is stored compactly in a two-dimensional array *ab* with *kl*+*ku*+1 rows and *n* columns. Columns of the matrix are stored in the corresponding columns of the array, and *diagonals* of the matrix are stored in rows of the array.

In Chapters 4 and 5, arrays that hold matrices in packed storage have names ending in *p*; arrays with matrices in band storage have names ending in *b*.

For more information on matrix storage schemes, see "Matrix Arguments" in Appendix B.

Mathematical Notation

Descriptions of LAPACK routines use the following notation:

$Ax = b$	A system of linear equations with an n by n matrix $A = \{a_{ij}\}$, a right-hand side vector $b = \{b_i\}$, and an unknown vector $x = \{x_i\}$.
$AX = B$	A set of systems with a common matrix A and multiple right-hand sides. The columns of B are individual right-hand sides, and the columns of X are the corresponding solutions.
$\|x\|$	the vector with elements $\|x_i\|$ (absolute values of x_i).
$\|A\|$	the matrix with elements $\|a_{ij}\|$ (absolute values of a_{ij}).
$\|\|x\|\|_\infty = \max_i \|x_i\|$	The *infinity-norm* of the vector x.
$\|\|A\|\|_\infty = \max_i \Sigma_j \|a_{ij}\|$	The *infinity-norm* of the matrix A.
$\|\|A\|\|_1 = \max_j \Sigma_i \|a_{ij}\|$	The *one-norm* of the matrix A. $\|\|A\|\|_1 = \|\|A^T\|\|_\infty = \|\|A^H\|\|_\infty$
$\kappa(A) = \|\|A\|\| \ \|\|A^{-1}\|\|$	The *condition number* of the matrix A.

Error Analysis

In practice, most computations are performed with rounding errors. Besides, you often need to solve a system $Ax = b$ where the data (the elements of A and b) are not known exactly. Therefore, it's important to understand how the data errors and rounding errors can affect the solution x.

Data perturbations. If x is the exact solution of $Ax = b$, and $x + \delta x$ is the exact solution of a perturbed problem $(A + \delta A)x = (b + \delta b)$, then

$$\frac{\|\delta x\|}{\|x\|} \le \kappa(A)\left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|}\right), \text{ where } \kappa(A) = \|A\|\|A^{-1}\|.$$

In other words, relative errors in A or b may be amplified in the solution vector x by a factor $\kappa(A) = \|A\| \|A^{-1}\|$ called the *condition number* of A.

Rounding errors have the same effect as relative perturbations $c(n)\varepsilon$ in the original data. Here ε is the *machine precision*, and $c(n)$ is a modest function of the matrix order n. The corresponding solution error is
$\|\delta x\|/\|x\| \le c(n)\kappa(A)\varepsilon$. (The value of $c(n)$ is seldom greater than $10n$.)

Thus, if your matrix A is *ill-conditioned* (that is, its condition number $\kappa(A)$ is very large), then the error in the solution x is also large; you may even encounter a complete loss of precision. LAPACK provides routines that allow you to estimate $\kappa(A)$ (see Routines for Estimating the Condition Number) and also give you a more precise estimate for the actual solution error (see Refining the Solution and Estimating Its Error).

Computational Routines

Table 3-1 lists the LAPACK computational routines for factorizing, equilibrating, and inverting *real* matrices, estimating their condition numbers, solving systems of equations with real matrices, refining the solution, and estimating its error.
Table 3-2 lists similar routines for *complex* matrices.

Table 3-1 Computational Routines for Systems of Equations with Real Matrices

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general	?getrf	?geequ	?getrs	?gecon	?gerfs	?getri
general band	?gbtrf	?gbequ	?gbtrs	?gbcon	?gbrfs	
general tridiagonal	?gttrf		?gttrs	?gtcon	?gtrfs	
symmetric positive-definite	?potrf	?poequ	?potrs	?pocon	?porfs	?potri
symmetric positive-definite, packed storage	?pptrf	?ppequ	?pptrs	?ppcon	?pprfs	?pptri
symmetric positive-definite, band	?pbtrf	?pbequ	?pbtrs	?pbcon	?pbrfs	
symmetric positive-definite, tridiagonal	?pttrf		?pttrs	?ptcon	?ptrfs	
symmetric indefinite	?sytrf		?sytrs	?sycon	?syrfs	?sytri
symmetric indefinite, packed storage	?sptrf		?sptrs	?spcon	?sprfs	?sptri
triangular			?trtrs	?trcon	?trrfs	?trtri
triangular, packed storage			?tptrs	?tpcon	?tprfs	?tptri
triangular band			?tbtrs	?tbcon	?tbrfs	

In this table **?** denotes **s** (single precision) or **d** (double precision).

Table 3-2 Computational Routines for Systems of Equations with Complex Matrices

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general	?getrf	?geequ	?getrs	?gecon	?gerfs	?getri
general band	?gbtrf	?gbequ	?gbtrs	?gbcon	?gbrfs	
general tridiagonal	?gttrf		?gttrs	?gtcon	?gtrfs	
Hermitian positive-definite	?potrf	?poequ	?potrs	?pocon	?porfs	?potri
Hermitian positive-definite, packed storage	?pptrf	?ppequ	?pptrs	?ppcon	?pprfs	?pptri
Hermitian positive-definite, band	?pbtrf	?pbequ	?pbtrs	?pbcon	?pbrfs	
Hermitian positive-definite, tridiagonal	?pttrf		?pttrs	?ptcon	?ptrfs	
Hermitian indefinite	?hetrf		?hetrs	?hecon	?herfs	?hetri
symmetric indefinite	?sytrf		?sytrs	?sycon	?syrfs	?sytri
Hermitian indefinite, packed storage	?hptrf		?hptrs	?hpcon	?hprfs	?hptri
symmetric indefinite, packed storage	?sptrf		?sptrs	?spcon	?sprfs	?sptri
triangular			?trtrs	?trcon	?trrfs	?trtri
triangular, packed storage			?tptrs	?tpcon	?tprfs	?tptri
triangular band			?tbtrs	?tbcon	?tbrfs	

In this table **?** stands for **c** (single precision complex) or **z** (double precision complex).

Routines for Matrix Factorization

This section describes the LAPACK routines for matrix factorization. The following factorizations are supported:

- *LU* factorization
- Cholesky factorization of real symmetric positive-definite matrices
- Cholesky factorization of Hermitian positive-definite matrices
- Bunch-Kaufman factorization of real and complex symmetric matrices
- Bunch-Kaufman factorization of Hermitian matrices.

You can compute the *LU* factorization using full and band storage of matrices; the Cholesky factorization using full, packed, and band storage; and the Bunch-Kaufman factorization using full and packed storage.

?getrf

Computes the LU factorization of a general m by n matrix.

Syntax

```
call sgetrf ( m, n, a, lda, ipiv, info )
call dgetrf ( m, n, a, lda, ipiv, info )
call cgetrf ( m, n, a, lda, ipiv, info )
call zgetrf ( m, n, a, lda, ipiv, info )
```

Description

The routine forms the *LU* factorization of a general m by n matrix A as

$$A = PLU,$$

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$). Usually A is square ($m = n$), and both L and U are triangular. The routine uses partial pivoting, with row interchanges.

Input Parameters

m	`INTEGER`. The number of rows in the matrix *A* ($m \geq 0$).
n	`INTEGER`. The number of columns in *A* ($n \geq 0$).
a	`REAL` for `sgetrf` `DOUBLE PRECISION` for `dgetrf` `COMPLEX` for `cgetrf` `DOUBLE COMPLEX` for `zgetrf`. Array, `DIMENSION` (*lda*, *). Contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*).
lda	`INTEGER`. The first dimension of *a*.

Output Parameters

a	Overwritten by *L* and *U*. The unit diagonal elements of *L* are not stored.
ipiv	`INTEGER`. Array, `DIMENSION` at least max(1,min(*m*,*n*)). The pivot indices: row *i* was interchanged with row *ipiv*(*i*).
info	`INTEGER`. If *info*=0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

Application Notes

The computed *L* and *U* are the exact factors of a perturbed matrix *A* + *E*, where

$$|E| \leq c(\min(m, n))\varepsilon P|L||U|$$

c(*n*) is a modest linear function of *n*, and ε is the machine precision.

The approximate number of floating-point operations for real flavors is

$(2/3)n^3$	if $m = n$,
$(1/3)n^2(3m-n)$	if $m > n$,
$(1/3)m^2(3n-m)$	if $m < n$.

The number of operations for complex flavors is 4 times greater.

After calling this routine with $m = n$, you can call the following:

?getrs	to solve $AX = B$ or $A^TX = B$ or $A^HX = B$;
?gecon	to estimate the condition number of A;
?getri	to compute the inverse of A.

?gbtrf

Computes the LU factorization of a general m by n band matrix.

Syntax

```
call sgbtrf ( m, n, kl, ku, ab, ldab, ipiv, info )
call dgbtrf ( m, n, kl, ku, ab, ldab, ipiv, info )
call cgbtrf ( m, n, kl, ku, ab, ldab, ipiv, info )
call zgbtrf ( m, n, kl, ku, ab, ldab, ipiv, info )
```

Description

The routine forms the LU factorization of a general m by n band matrix A with kl non-zero sub-diagonals and ku non-zero super-diagonals. Usually A is square ($m = n$), and then

$$A = PLU$$

where P is a permutation matrix; L is lower triangular with unit diagonal elements and at most kl non-zero elements in each column; U is an upper triangular band matrix with $kl + ku$ super-diagonals. The routine uses partial pivoting, with row interchanges (which creates the additional kl super-diagonals in U).

Input Parameters

m	`INTEGER`. The number of rows in the matrix A ($m \geq 0$).
n	`INTEGER`. The number of columns in A ($n \geq 0$).
kl	`INTEGER`. The number of sub-diagonals within the band of A ($kl \geq 0$).
ku	`INTEGER`. The number of super-diagonals within the band of A ($ku \geq 0$).
ab	`REAL` for `sgbtrf` `DOUBLE PRECISION` for `dgbtrf` `COMPLEX` for `cgbtrf`

	`DOUBLE COMPLEX` for `zgbtrf`. Array, `DIMENSION` (*ldab*, *). The array *ab* contains the matrix *A* in band storage (see Matrix Storage Schemes). The second dimension of *ab* must be at least max(1, *n*).
ldab	`INTEGER`. The first dimension of the array *ab*. ($ldab \geq 2kl + ku + 1$)

Output Parameters

ab	Overwritten by *L* and *U*. The diagonal and *kl* + *ku* super-diagonals of *U* are stored in the first 1 + *kl* + *ku* rows of *ab*. The multipliers used to form *L* are stored in the next *kl* rows.
ipiv	`INTEGER`. Array, `DIMENSION` at least max(1,min(*m*,*n*)). The pivot indices: row *i* was interchanged with row *ipiv*(*i*).
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

Application Notes

The computed *L* and *U* are the exact factors of a perturbed matrix *A* + *E*, where

$$|E| \leq c(kl + ku + 1)\varepsilon P|L||U|$$

$c(k)$ is a modest linear function of *k*, and ε is the machine precision.

The total number of floating-point operations for real flavors varies between approximately $2n(ku+1)kl$ and $2n(kl+ku+1)kl$. The number of operations for complex flavors is 4 times greater. All these estimates assume that *kl* and *ku* are much less than min(*m*, *n*).

After calling this routine with *m* = *n*, you can call the following:

?gbtrs	to solve $AX = B$ or $A^TX = B$ or $A^HX = B$;
?gbcon	to estimate the condition number of *A*.

?gttrf

Computes the LU factorization of a tridiagonal matrix.

Syntax

```
call sgttrf ( n, dl, d, du, du2, ipiv, info )
call dgttrf ( n, dl, d, du, du2, ipiv, info )
call cgttrf ( n, dl, d, du, du2, ipiv, info )
call zgttrf ( n, dl, d, du, du2, ipiv, info )
```

Description

The routine computes the *LU* factorization of a real or complex tridiagonal matrix *A* in the form

$$A = PLU,$$

where *P* is a permutation matrix; *L* is lower bidiagonal with unit diagonal elements; and *U* is an upper triangular matrix with nonzeroes in only the main diagonal and first two superdiagonals. The routine uses elimination with partial pivoting and row interchanges .

Input Parameters

n — `INTEGER`. The order of the matrix *A* ($n \geq 0$).

dl, d, du — `REAL` for `sgttrf`
`DOUBLE PRECISION` for `dgttrf`
`COMPLEX` for `cgttrf`
`DOUBLE COMPLEX` for `zgttrf`.
Arrays containing elements of A.
The array *dl* of dimension (n - 1) contains the sub-diagonal elements of *A*.
The array *d* of dimension n contains the diagonal elements of *A*.
The array *du* of dimension (n - 1) contains the super-diagonal elements of *A*.

Output Parameters

dl — Overwritten by the (n-1) multipliers that define the matrix *L* from the *LU* factorization of A.

d — Overwritten by the n diagonal elements of the upper triangular matrix *U* from the *LU* factorization of A.

du — Overwritten by the (n-1) elements of the first super-diagonal of *U*.

du2	`REAL` for `sgttrf` `DOUBLE PRECISION` for `dgttrf` `COMPLEX` for `cgttrf` `DOUBLE COMPLEX` for `zgttrf`. Array, dimension (*n*-2). On exit, *du2* contains (*n*-2) elements of the second super-diagonal of *U*.
ipiv	`INTEGER`. Array, dimension (*n*). The pivot indices: row *i* was interchanged with row *ipiv*(*i*).
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by zero will occur if you use the factor *U* for solving a system of linear equations.

Application Notes

`?gbtrs`	to solve $AX = B$ or $A^TX = B$ or $A^HX = B$;
`?gbcon`	to estimate the condition number of *A*.

?potrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix.

Syntax

```
call spotrf ( uplo, n, a, lda, info )
call dpotrf ( uplo, n, a, lda, info )
call cpotrf ( uplo, n, a, lda, info )
call zpotrf ( uplo, n, a, lda, info )
```

Description

This routine forms the Cholesky factorization of a symmetric positive- definite or, for complex data, Hermitian positive-definite matrix *A*:

$A = U^HU$ if *uplo*='U'

$A = LL^H$ if `uplo='L'`,

where L is a lower triangular matrix and U is upper triangular.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates whether the upper or lower triangular part of A is stored and how A is factored:
If *uplo* = `'U'`, the array *a* stores the upper triangular part of the matrix A, and A is factored as $U^H U$.
If *uplo* = `'L'`, the array *a* stores the lower triangular part of the matrix A; A is factored as LL^H.

n — `INTEGER`. The order of matrix A ($n \geq 0$).

a — `REAL` for `spotrf`
`DOUBLE PRECISION` for `dpotrf`
`COMPLEX` for `cpotrf`
`DOUBLE COMPLEX` for `zpotrf`.
Array, `DIMENSION` (*lda*, *).
The array *a* contains either the upper or the lower triangular part of the matrix A (see *uplo*).
The second dimension of *a* must be at least max(1, *n*).

lda — `INTEGER`. The first dimension of *a*.

Output Parameters

a — The upper or lower triangular part of *a* is overwritten by the Cholesky factor U or L, as specified by *uplo*.

info — `INTEGER`. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, the leading minor of order *i* (and hence the matrix A itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix A.

Application Notes

If *uplo* = `'U'`, the computed factor U is the exact factor of a perturbed matrix $A + E$, where

$$|E| \leq c(n)\varepsilon |U^H||U|, \quad |e_{ij}| \leq c(n)\varepsilon \sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

A similar estimate holds for *uplo* = `'L'`.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following:

?potrs	to solve $AX = B$;
?pocon	to estimate the condition number of A;
?potri	to compute the inverse of A.

?pptrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix using packed storage.

Syntax

```
call spptrf ( uplo, n, ap, info )
call dpptrf ( uplo, n, ap, info )
call cpptrf ( uplo, n, ap, info )
call zpptrf ( uplo, n, ap, info )
```

Description

This routine forms the Cholesky factorization of a symmetric positive- definite or, for complex data, Hermitian positive-definite packed matrix A:

$A = U^H U$ if `uplo='U'`

$A = LL^H$ if `uplo='L'`

where L is a lower triangular matrix and U is upper triangular.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates whether the upper or lower triangular part of A is packed in the array *ap*, and how A is factored:

If `uplo` = 'U', the array `ap` stores the upper triangular part of the matrix A, and A is factored as $U^H U$.
If `uplo` = 'L', the array `ap` stores the lower triangular part of the matrix A; A is factored as LL^H.

n — INTEGER. The order of matrix A ($n \ge 0$).

ap — REAL for spptrf
DOUBLE PRECISION for dpptrf
COMPLEX for cpptrf
DOUBLE COMPLEX for zpptrf.
Array, DIMENSION at least max(1,n(n+1)/2).
The array `ap` contains either the upper or the lower triangular part of the matrix A (as specified by `uplo`) in *packed storage* (see Matrix Storage Schemes).

Output Parameters

ap — The upper or lower triangular part of A in packed storage is overwritten by the Cholesky factor U or L, as specified by `uplo`.

info — INTEGER. If `info`=0, the execution is successful.
If `info` = `-i`, the `i`th parameter had an illegal value.
If `info` = `i`, the leading minor of order `i` (and hence the matrix A itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix A.

Application Notes

If `uplo` = 'U', the computed factor U is the exact factor of a perturbed matrix $A + E$, where

$$|E| \le c(n)\varepsilon\,|U^H||U|, \quad |e_{ij}| \le c(n)\varepsilon\sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

A similar estimate holds for `uplo` = 'L'.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following:

?pptrs	to solve $AX = B$;
?ppcon	to estimate the condition number of A;
?pptri	to compute the inverse of A.

?pbtrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite band matrix.

Syntax

```
call spbtrf ( uplo, n, kd, ab, ldab, info )
call dpbtrf ( uplo, n, kd, ab, ldab, info )
call cpbtrf ( uplo, n, kd, ab, ldab, info )
call zpbtrf ( uplo, n, kd, ab, ldab, info )
```

Description

This routine forms the Cholesky factorization of a symmetric positive- definite or, for complex data, Hermitian positive-definite band matrix A:

$A = U^H U$ if *uplo*='U'

$A = LL^H$ if *uplo*='L'

where L is a lower triangular matrix and U is upper triangular.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored in the array *ab*, and how A is factored: If *uplo* = 'U', the array *ab* stores the upper triangular part of the matrix A, and A is factored as $U^H U$. If *uplo* = 'L', the array *ab* stores the lower triangular part of the matrix A; A is factored as LL^H.
n	INTEGER. The order of matrix A ($n \geq 0$).
kd	INTEGER. The number of super-diagonals or sub-diagonals in the matrix A ($kd \geq 0$).
ab	REAL for spbtrf DOUBLE PRECISION for dpbtrf COMPLEX for cpbtrf DOUBLE COMPLEX for zpbtrf. Array, DIMENSION (*ldab*,*).

The array *ap* contains either the upper or the lower triangular part of the matrix *A* (as specified by *uplo*) in *band storage* (see Matrix Storage Schemes). The second dimension of *ab* must be at least max(1, *n*).

ldab INTEGER. The first dimension of the array *ab*. ($ldab \geq kd + 1$)

Output Parameters

ap The upper or lower triangular part of *A* (in band storage) is overwritten by the Cholesky factor *U* or *L*, as specified by *uplo*.

info INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, the leading minor of order *i* (and hence the matrix *A* itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix *A*.

Application Notes

If *uplo* = 'U', the computed factor *U* is the exact factor of a perturbed matrix *A* + *E*, where

$$|E| \leq c(kd+1)\varepsilon\,|U^H||U|, \quad |e_{ij}| \leq c(kd+1)\varepsilon\,\sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

A similar estimate holds for *uplo* = 'L'.

The total number of floating-point operations for real flavors is approximately $n(kd+1)^2$. The number of operations for complex flavors is 4 times greater. All these estimates assume that *kd* is much less than *n*.

After calling this routine, you can call the following:

?pbtrs to solve $AX = B$;

?pbcon to estimate the condition number of *A*;

?pttrf

Computes the factorization of a symmetric (Hermitian) positive-definite tridiagonal matrix.

Syntax

```
call spttrf ( n, d, e, info )
call dpttrf ( n, d, e, info )
call cpttrf ( n, d, e, info )
call zpttrf ( n, d, e, info )
```

Description

This routine forms the factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite tridiagonal matrix A:

$A = LDL^H$, where D is diagonal and L is unit lower bidiagonal. The factorization may also be regarded as having the form $A = U^H DU$, where D is unit upper bidiagonal.

Input Parameters

n — `INTEGER`. The order of the matrix A ($n \geq 0$).

d — `REAL` for `spttrf, cpttrf`
`DOUBLE PRECISION` for `dpttrf, zpttrf`.
Array, dimension (*n*). Contains the diagonal elements of A.

e — `REAL` for `spttrf`
`DOUBLE PRECISION` for `dpttrf`
`COMPLEX` for `cpttrf`
`DOUBLE COMPLEX` for `zpttrf`.
Array, dimension (n - 1). Contains the sub-diagonal elements of A.

Output Parameters

d — Overwritten by the *n* diagonal elements of the diagonal matrix D from the LDL^H factorization of A.

e — Overwritten by the (n - 1) off-diagonal elements of the unit bidiagonal factor L or U from the factorization of A.

info INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, the leading minor of order *i* (and hence the matrix *A* itself) is not positive-definite; if $i < n$, the factorization could not be completed, while if $i = n$, the factorization was completed, but $d(n) = 0$.

?sytrf

Computes the Bunch-Kaufman factorization of a symmetric matrix.

Syntax

```
call ssytrf ( uplo, n, a, lda, ipiv, work, lwork, info )
call dsytrf ( uplo, n, a, lda, ipiv, work, lwork, info )
call csytrf ( uplo, n, a, lda, ipiv, work, lwork, info )
call zsytrf ( uplo, n, a, lda, ipiv, work, lwork, info )
```

Description

This routine forms the Bunch-Kaufman factorization of a symmetric matrix:

if *uplo*='U', $A = PUDU^T P^T$

if *uplo*='L', $A = PLDL^T P^T$

where *A* is the input matrix, *P* is a permutation matrix, *U* and *L* are upper and lower triangular matrices with unit diagonal, and *D* is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. *U* and *L* have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of *D*.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of *A* is stored and how *A* is factored:
If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix *A*, and *A* is factored as $PUDU^T P^T$.
If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix *A*; *A* is factored as $PLDL^T P^T$.

n INTEGER. The order of matrix *A* ($n \geq 0$).

a	`REAL` for `ssytrf` `DOUBLE PRECISION` for `dsytrf` `COMPLEX` for `csytrf` `DOUBLE COMPLEX` for `zsytrf`. Array, `DIMENSION` (*lda*, *). The array *a* contains either the upper or the lower triangular part of the matrix *A* (see *uplo*). The second dimension of *a* must be at least max(1, *n*).
lda	`INTEGER`. The first dimension of *a*; at least max(1, *n*).
work	Same type as *a*. Workspace array of dimension *lwork*
lwork	`INTEGER`. The size of the *work* array (*lwork* ≥ *n*) See Application notes for the suggested value of *lwork*.

Output Parameters

a	The upper or lower triangular part of *a* is overwritten by details of the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*).
work(1)	If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
ipiv	`INTEGER`. Array, `DIMENSION` at least max(1,*n*). Contains details of the interchanges and the block structure of *D*. If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 block, and the *i*th row and column of *A* was interchanged with the *k*th row and column. If *uplo* = 'U' and *ipiv*(*i*) =*ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*-1, and (*i*-1)th row and column of *A* was interchanged with the *m*th row and column. If *uplo* = 'L' and *ipiv*(*i*) =*ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)th row and column of *A* was interchanged with the *m*th row and column.
info	`INTEGER`. If *info*=0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular. Division by 0 will occur if you use *D* for solving a system of linear equations.

Application Notes

For better performance, try using *lwork* =*n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The 2-by-2 unit diagonal blocks and the unit diagonal elements of *U* and *L* are not stored. The remaining elements of *U* and *L* are stored in the corresponding columns of the array *a*, but additional row interchanges are required to recover *U* or *L* explicitly (which is seldom necessary).

If *ipiv*(*i*) = *i* for all *i* =1...*n*, then all off-diagonal elements of *U* (*L*) are stored explicitly in the corresponding elements of the array *a*.

If *uplo* = 'U', the computed factors *U* and *D* are the exact factors of a perturbed matrix *A* + *E*, where

$$|E| \le c(n)\varepsilon P|U||D||U^T|P^T$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.
A similar estimate holds for the computed *L* and *D* when *uplo* = 'L'.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following:

?sytrs	to solve $AX = B$;
?sycon	to estimate the condition number of A;
?sytri	to compute the inverse of A.

?hetrf

Computes the Bunch-Kaufman factorization of a complex Hermitian matrix.

Syntax

```
call chetrf ( uplo, n, a, lda, ipiv, work, lwork, info )
call zhetrf ( uplo, n, a, lda, ipiv, work, lwork, info )
```

Description

This routine forms the Bunch-Kaufman factorization of a Hermitian matrix:

if `uplo='U'`, $A = PUDU^H P^T$

if `uplo='L'`, $A = PLDL^H P^T$

where A is the input matrix, P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates whether the upper or lower triangular part of A is stored and how A is factored:
If *uplo* = `'U'`, the array *a* stores the upper triangular part of the matrix A, and A is factored as $PUDU^H P^T$.
If *uplo* = `'L'`, the array *a* stores the lower triangular part of the matrix A; A is factored as $PLDL^H P^T$.

n — `INTEGER`. The order of matrix A ($n \geq 0$).

a — `COMPLEX` for `chetrf`
`DOUBLE COMPLEX` for `zhetrf`.
Array, `DIMENSION` (*lda*, *).
The array *a* contains either the upper or the lower triangular part of the matrix A (see *uplo*).
The second dimension of *a* must be at least max(1, *n*).

lda — `INTEGER`. The first dimension of *a*; at least max(1, *n*).

work Same type as *a*. Workspace array of dimension *lwork*

lwork INTEGER. The size of the *work* array ($lwork \geq n$)
See Application notes for the suggested value of *lwork*.

Output Parameters

a The upper or lower triangular part of *a* is overwritten by details of the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*).

work(1) If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

ipiv INTEGER.
Array, DIMENSION at least max(1,*n*).
Contains details of the interchanges and the block structure of *D*.
If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 block, and the *i*th row and column of *A* was interchanged with the *k*th row and column.

If *uplo* = 'U' and $ipiv(i) = ipiv(i-1) = -m < 0$, then *D* has a 2-by-2 block in rows/columns *i* and *i*-1, and (*i*-1)th row and column of *A* was interchanged with the *m*th row and column.

If *uplo* = 'L' and $ipiv(i) = ipiv(i+1) = -m < 0$, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)th row and column of *A* was interchanged with the *m*th row and column.

info INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular. Division by 0 will occur if you use *D* for solving a system of linear equations.

Application Notes

This routine is suitable for Hermitian matrices that are not known to be positive-definite. If *A* is in fact positive-definite, the routine does not perform interchanges, and no 2-by-2 diagonal blocks occur in *D*.

For better performance, try using *lwork* =*n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The 2-by-2 unit diagonal blocks and the unit diagonal elements of U and L are not stored. The remaining elements of U and L are stored in the corresponding columns of the array `a`, but additional row interchanges are required to recover U or L explicitly (which is seldom necessary).

If `ipiv(i)` = `i` for all `i` =1...`n`, then all off-diagonal elements of U (L) are stored explicitly in the corresponding elements of the array `a`.

If `uplo` = `'U'`, the computed factors U and D are the exact factors of a perturbed matrix $A + E$, where

$$|E| \le c(n)\varepsilon P|U||D||U^T|P^T$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.
A similar estimate holds for the computed L and D when `uplo` = `'L'`.

The total number of floating-point operations is approximately $(4/3)n^3$.

After calling this routine, you can call the following:

?hetrs	to solve $AX = B$;
?hecon	to estimate the condition number of A;
?hetri	to compute the inverse of A.

?sptrf

Computes the Bunch-Kaufman factorization of a symmetric matrix using packed storage.

Syntax

```
call ssptrf ( uplo, n, ap, ipiv, info )
call dsptrf ( uplo, n, ap, ipiv, info )
call csptrf ( uplo, n, ap, ipiv, info )
call zsptrf ( uplo, n, ap, ipiv, info )
```

Description

This routine forms the Bunch-Kaufman factorization of a symmetric matrix A using packed storage:

if `uplo`=`'U'`, $A = PUDU^TP^T$

if `uplo='L'`, $A = PLDL^TP^T$

where P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Indicates whether the upper or lower triangular part of A is packed in the array *ap* and how A is factored: If *uplo* = `'U'`, the array *ap* stores the upper triangular part of the matrix A, and A is factored as $PUDU^TP^T$. If *uplo* = `'L'`, the array *ap* stores the lower triangular part of the matrix A; A is factored as $PLDL^TP^T$.
n	`INTEGER`. The order of matrix A ($n \ge 0$).
ap	`REAL` for `ssptrf` `DOUBLE PRECISION` for `dsptrf` `COMPLEX` for `csptrf` `DOUBLE COMPLEX` for `zsptrf`. Array, `DIMENSION` at least max(1,n(n+1)/2). The array *ap* contains either the upper or the lower triangular part of the matrix A (as specified by *uplo*) in *packed storage* (see Matrix Storage Schemes).

Output Parameters

ap	The upper or lower triangle of A (as specified by *uplo*) is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).
ipiv	`INTEGER`. Array, `DIMENSION` at least max(1,n). Contains details of the interchanges and the block structure of D. If *ipiv*(*i*) = $k > 0$, then d_{ii} is a 1-by-1 block, and the *i*th row and column of A was interchanged with the *k*th row and column. If *uplo* = `'U'` and *ipiv*(*i*) =*ipiv*(*i*-1) = -*m* < 0, then D has a 2-by-2 block in rows/columns *i* and *i*-1, and (*i*-1)th row and column of A was interchanged with the *m*th row and column.

If *uplo* = 'L' and *ipiv*(*i*) =*ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)th row and column of *A* was interchanged with the *m*th row and column.

info — INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular. Division by 0 will occur if you use *D* for solving a system of linear equations.

Application Notes

The 2-by-2 unit diagonal blocks and the unit diagonal elements of *U* and *L* are not stored. The remaining elements of *U* and *L* overwrite elements of the corresponding columns of the matrix *A*, but additional row interchanges are required to recover *U* or *L* explicitly (which is seldom necessary).

If *ipiv*(*i*) = *i* for all *i* =1...*n*, then all off-diagonal elements of *U* (*L*) are stored explicitly in packed form.

If *uplo* = 'U', the computed factors *U* and *D* are the exact factors of a perturbed matrix *A* + *E*, where

$$|E| \le c(n)\varepsilon P|U||D||U^T|P^T$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.
A similar estimate holds for the computed *L* and *D* when *uplo* = 'L'.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following:

?sptrs	to solve $AX = B$;
?spcon	to estimate the condition number of *A*;
?sptri	to compute the inverse of *A*.

?hptrf

Computes the Bunch-Kaufman factorization of a complex Hermitian matrix using packed storage.

Syntax

```
call chptrf ( uplo, n, ap, ipiv, info )
call zhptrf ( uplo, n, ap, ipiv, info )
```

Description

This routine forms the Bunch-Kaufman factorization of a Hermitian matrix using packed storage:

if `uplo='U'`, $A = PUDU^HP^T$

if `uplo='L'`, $A = PLDL^HP^T$

where A is the input matrix, P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates whether the upper or lower triangular part of A is packed and how A is factored:

If *uplo* = `'U'`, the array *ap* stores the upper triangular part of the matrix A, and A is factored as $PUDU^HP^T$.
If *uplo* = `'L'`, the array *ap* stores the lower triangular part of the matrix A; A is factored as $PLDL^HP^T$.

n — `INTEGER`. The order of matrix A ($n \ge 0$).

ap — `COMPLEX` for `chptrf`
`DOUBLE COMPLEX` for `zhptrf`.
Array, `DIMENSION` at least max(1,$n(n+1)/2$).
The array *ap* contains either the upper or the lower triangular part of the matrix A (as specified by *uplo*) in *packed storage* (see Matrix Storage Schemes).

Output Parameters

ap The upper or lower triangle of A (as specified by *uplo*) is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).

ipiv INTEGER.
Array, DIMENSION at least max(1,*n*).
Contains details of the interchanges and the block structure of D.
If *ipiv*(*i*) = $k > 0$, then d_{ii} is a 1-by-1 block, and the *i*th row and column of A was interchanged with the *k*th row and column.

If *uplo* = 'U' and *ipiv*(*i*) =*ipiv*(*i*-1) = -$m < 0$, then D has a 2-by-2 block in rows/columns *i* and *i*-1, and (*i*-1)th row and column of A was interchanged with the *m*th row and column.

If *uplo* = 'L' and *ipiv*(*i*) =*ipiv*(*i*+1) = -$m < 0$, then D has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)th row and column of A was interchanged with the *m*th row and column.

info INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

Application Notes

The 2-by-2 unit diagonal blocks and the unit diagonal elements of U and L are not stored. The remaining elements of U and L are stored in the corresponding columns of the array *a*, but additional row interchanges are required to recover U or L explicitly (which is seldom necessary).

If *ipiv*(*i*) = *i* for all *i* =1...*n*, then all off-diagonal elements of U (L) are stored explicitly in the corresponding elements of the array *a*.

If *uplo* = 'U', the computed factors U and D are the exact factors of a perturbed matrix $A + E$, where

$$|E| \le c(n)\varepsilon P|U||D||U^T|P^T$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.
A similar estimate holds for the computed L and D when *uplo* = 'L'.

The total number of floating-point operations is approximately $(4/3)n^3$.

After calling this routine, you can call the following:

?hptrs	to solve $AX = B$;
?hpcon	to estimate the condition number of A;
?hptri	to compute the inverse of A.

Routines for Solving Systems of Linear Equations

This section describes the LAPACK routines for solving systems of linear equations. Before calling most of these routines, you need to factorize the matrix of your system of equations (see Routines for Matrix Factorization in this chapter). However, the factorization is not necessary if your system of equations has a triangular matrix.

?getrs

Solves a system of linear equations with an LU-factored square matrix, with multiple right-hand sides.

Syntax

```
call sgetrs (trans, n, nrhs, a, lda, ipiv, b, ldb, info)
call dgetrs (trans, n, nrhs, a, lda, ipiv, b, ldb, info)
call cgetrs (trans, n, nrhs, a, lda, ipiv, b, ldb, info)
call zgetrs (trans, n, nrhs, a, lda, ipiv, b, ldb, info)
```

Description

This routine solves for X the following systems of linear equations:

$AX = B$	if *trans*='N',
$A^T X = B$	if *trans*='T',
$A^H X = B$	if *trans*='C' (for complex matrices only).

Before calling this routine, you must call ?getrf to compute the LU factorization of A.

Input Parameters

trans CHARACTER*1. Must be 'N' or 'T' or 'C'.
Indicates the form of the equations:
If *trans* = 'N', then $AX = B$ is solved for X.

If *trans* = 'T', then $A^T X = B$ is solved for X.
If *trans* = 'C', then $A^H X = B$ is solved for X.

n — INTEGER. The order of A; the number of rows in B ($n \geq 0$).

nrhs — INTEGER. The number of right-hand sides ($nrhs \geq 0$).

a, b — REAL for sgetrs
DOUBLE PRECISION for dgetrs
COMPLEX for cgetrs
DOUBLE COMPLEX for zgetrs.
Arrays: a(*lda*,*), b(*ldb*,*).

The array *a* contains the matrix A.
The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations.

The second dimension of *a* must be at least max(1,*n*), the second dimension of *b* at least max(1,*nrhs*).

lda — INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).

ldb — INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

ipiv — INTEGER.
Array, DIMENSION at least max(1,*n*).
The *ipiv* array, as returned by ?getrf.

Output Parameters

b — Overwritten by the solution matrix X.

info — INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For each right-hand side b, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$ where

$$|E| \leq c(n)\varepsilon P|L||U|$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(n)\ \mathrm{cond}\ (A, x)\varepsilon$$

where $\mathrm{cond}(A,x) = \|\,|A^{-1}|\,|A|\,|x|\,\|_\infty / \|x\|_\infty \le \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that cond(A,x) can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number κ_∞ (A), call `?gecon`.
To refine the solution and estimate the error, call `?gerfs`.

?gbtrs

Solves a system of linear equations with an LU-factored band matrix, with multiple right-hand sides.

Syntax

```
call sgbtrs (trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info)
call dgbtrs (trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info)
call cgbtrs (trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info)
call zgbtrs (trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info)
```

Description

This routine solves for X the following systems of linear equations:

$AX = B$	if *trans*=`'N'`,
$A^T X = B$	if *trans*=`'T'`,
$A^H X = B$	if *trans*=`'C'` (for complex matrices only).

Here A is an LU-factored general band matrix of order *n* with *kl* non-zero sub-diagonals and *ku* non-zero super-diagonals. Before calling this routine, you must call `?gbtrf` to compute the LU factorization of A.

Input Parameters

trans	`CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`.
n	`INTEGER`. The order of *A*; the number of rows in *B* ($n \geq 0$).
kl	`INTEGER`. The number of sub-diagonals within the band of *A* ($kl \geq 0$).
ku	`INTEGER`. The number of super-diagonals within the band of *A* ($ku \geq 0$).
nrhs	`INTEGER`. The number of right-hand sides ($nrhs \geq 0$).
ab, b	`REAL` for `sgbtrs` `DOUBLE PRECISION` for `dgbtrs` `COMPLEX` for `cgbtrs` `DOUBLE COMPLEX` for `zgbtrs`. Arrays: *ab*(*ldab*,*), *b*(*ldb*,*). The array *ab* contains the matrix *A* in *band storage* (see Matrix Storage Schemes). The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *ab* must be at least max(1, *n*), the second dimension of *b* at least max(1,*nrhs*).
ldab	`INTEGER`. The first dimension of the array *ab*. ($ldab \geq 2kl + ku + 1$).
ldb	`INTEGER`. The first dimension of *b*; $ldb \geq \max(1, n)$.
ipiv	`INTEGER`. Array, `DIMENSION` at least max(1,*n*). The *ipiv* array, as returned by `?gbtrf`.

Output Parameters

b	Overwritten by the solution matrix *X*.
info	`INTEGER`. If *info*=0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(kl + ku + 1)\varepsilon P|L||U|$$

$c(k)$ is a modest linear function of *k*, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(kl + ku + 1)\ \mathrm{cond}\ (A,\ x)\varepsilon$$

where $\mathrm{cond}(A,x) = \|\ |A^{-1}|\ |A|\ |x|\ \|_\infty / \|x\|_\infty \le \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that cond(A,x) can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector is $2n(ku + 2kl)$ for real flavors. The number of operations for complex flavors is 4 times greater. All these estimates assume that `kl` and `ku` are much less than min(`m,n`).

To estimate the condition number $\kappa_\infty(A)$, call `?gbcon`.
To refine the solution and estimate the error, call `?gbrfs`.

?gttrs

Solves a system of linear equations with a tridiagonal matrix using the LU factorization computed by `?gttrf`.

Syntax

```
call sgttrs (trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info)
call dgttrs (trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info)
call cgttrs (trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info)
call zgttrs (trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info)
```

Description

This routine solves for X the following systems of linear equations with multiple right hand sides:

$AX = B$ if `trans='N'`,

$A^TX = B$ if `trans='T'`,

$A^HX = B$ if `trans='C'` (for complex matrices only).

Before calling this routine, you must call `?gttrf` to compute the LU factorization of A.

Input Parameters

trans — CHARACTER*1. Must be 'N' or 'T' or 'C'.
Indicates the form of the equations:
If *trans* = 'N', then $AX = B$ is solved for X.
If *trans* = 'T', then $A^TX = B$ is solved for X.
If *trans* = 'C', then $A^HX = B$ is solved for X.

n — INTEGER. The order of A ($n \geq 0$).

nrhs — INTEGER. The number of right-hand sides, i.e., the number of columns in B ($nrhs \geq 0$).

dl,d,du,du2,b — REAL for sgttrs
DOUBLE PRECISION for dgttrs
COMPLEX for cgttrs
DOUBLE COMPLEX for zgttrf.
Arrays: *dl*(n - 1), *d*(n), *du*(n - 1), *du2*(n - 2), *b*(*ldb,nrhs*).
The array *dl* contains the (n - 1) multipliers that define the matrix L from the LU factorization of A.
The array *d* contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A.
The array *du* contains the (n - 1) elements of the first super-diagonal of U.
The array *du2* contains the (n - 2) elements of the second super-diagonal of U.
The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations.

ldb — INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

ipiv — INTEGER.
Array, DIMENSION (n).
The *ipiv* array, as returned by ?gttrf.

Output Parameters

b — Overwritten by the solution matrix X.

info — INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For each right-hand side b, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$ where

$$|E| \le c(n)\varepsilon P|L||U|$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(n) \text{ cond } (A, x)\varepsilon$$

where $\text{cond}(A,x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \le \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that cond(A,x) can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call `?gecon`.
To refine the solution and estimate the error, call `?gerfs`.

?potrs

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite matrix.

Syntax

```
call spotrs ( uplo, n, nrhs, a, lda, b, ldb, info )
call dpotrs ( uplo, n, nrhs, a, lda, b, ldb, info )
call cpotrs ( uplo, n, nrhs, a, lda, b, ldb, info )
call zpotrs ( uplo, n, nrhs, a, lda, b, ldb, info )
```

Description

This routine solves for X the system of linear equations $AX = B$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A, given the Cholesky factorization of A:

$A = U^H U$ if `uplo` = `'U'`

$A = LL^H$ if `uplo` = `'L'`

where L is a lower triangular matrix and U is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix B.

Before calling this routine, you must call `?potrf` to compute the Cholesky factorization of A.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Indicates how the input matrix A has been factored: If *uplo* = `'U'`, the array *a* stores the factor U of the Cholesky factorization $A = U^H U$. If *uplo* = `'L'`, the array *a* stores the factor L of the Cholesky factorization $A = LL^H$.
n	`INTEGER`. The order of matrix A ($n \geq 0$).
nrhs	`INTEGER`. The number of right-hand sides ($nrhs \geq 0$).
a, b	`REAL` for `spotrs` `DOUBLE PRECISION` for `dpotrs` `COMPLEX` for `cpotrs` `DOUBLE COMPLEX` for `zpotrs`. Arrays: *a*(*lda*,*), *b*(*ldb*,*). The array *a* contains the factor U or L (see *uplo*). The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of *a* must be at least max(1,*n*), the second dimension of *b* at least max(1,*nrhs*).
lda	`INTEGER`. The first dimension of *a*; $lda \geq \max(1, n)$.
ldb	`INTEGER`. The first dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

b	Overwritten by the solution matrix X.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

If *uplo* = 'U', the computed solution for each right-hand side b is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \le c(n)\varepsilon |U^H||U|$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.
A similar estimate holds for *uplo* = 'L'.
If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(n)\ \mathrm{cond}(A, x)\varepsilon$$

where $\mathrm{cond}(A,x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \le \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that cond(A,x) can be much smaller than $\kappa_\infty(A)$.
The approximate number of floating-point operations for one right-hand side vector b is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call `?pocon`.
To refine the solution and estimate the error, call `?porfs`.

?pptrs

Solves a system of linear equations with a packed Cholesky-factored symmetric (Hermitian) positive-definite matrix.

Syntax

```
call spptrs ( uplo, n, nrhs, ap, b, ldb, info )
call dpptrs ( uplo, n, nrhs, ap, b, ldb, info )
call cpptrs ( uplo, n, nrhs, ap, b, ldb, info )
call zpptrs ( uplo, n, nrhs, ap, b, ldb, info )
```

Description

This routine solves for X the system of linear equations $AX = B$ with a packed symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A, given the Cholesky factorization of A:

$A = U^H U$ if `uplo = 'U'`

$A = LL^H$ if `uplo = 'L'`

where L is a lower triangular matrix and U is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix B.

Before calling this routine, you must call `?pptrf` to compute the Cholesky factorization of A.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates how the input matrix A has been factored:
If `uplo = 'U'`, the array *a* stores the packed factor U of the Cholesky factorization $A = U^H U$.
If `uplo = 'L'`, the array *a* stores the packed factor L of the Cholesky factorization $A = LL^H$.

n — `INTEGER`. The order of matrix A ($n \geq 0$).

nrhs — `INTEGER`. The number of right-hand sides ($nrhs \geq 0$).

ap, b — `REAL` for `spptrs`
`DOUBLE PRECISION` for `dpptrs`
`COMPLEX` for `cpptrs`
`DOUBLE COMPLEX` for `zpptrs`.
Arrays: *ap*(*), *b*(*ldb*,*)
The dimension of *ap* must be at least max(1,n(n+1)/2).
The array *ap* contains the factor U or L, as specified by *uplo*, in *packed storage* (see Matrix Storage Schemes).
The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

ldb — `INTEGER`. The first dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

b — Overwritten by the solution matrix X.

info — `INTEGER`. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

If *uplo* = 'U', the computed solution for each right-hand side *b* is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \le c(n)\varepsilon |U^H||U|$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

A similar estimate holds for *uplo* = 'L'.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(n)\ \mathrm{cond}(A, x)\varepsilon$$

where $\mathrm{cond}(A,x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \le \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that cond(A,x) can be much smaller than $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?ppcon.
To refine the solution and estimate the error, call ?pprfs.

?pbtrs

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite band matrix.

Syntax

```
call spbtrs (uplo, n, kd, nrhs, ab, ldab, b, ldb, info)
call dpbtrs (uplo, n, kd, nrhs, ab, ldab, b, ldb, info)
call cpbtrs (uplo, n, kd, nrhs, ab, ldab, b, ldb, info)
call zpbtrs (uplo, n, kd, nrhs, ab, ldab, b, ldb, info)
```

Description

This routine solves for X the system of linear equations $AX = B$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite ***band*** matrix A, given the Cholesky factorization of A:

$A = U^H U$	if `uplo='U'`
$A = LL^H$	if `uplo='L'`

where L is a lower triangular matrix and U is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix B.

Before calling this routine, you must call `?pbtrf` to compute the Cholesky factorization of A in the band storage form.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates how the input matrix A has been factored:
If *uplo* = `'U'`, the array *a* stores the factor U of the factorization $A = U^H U$ in the band storage form.
If *uplo* = `'L'`, the array *a* stores the factor L of the factorization $A = LL^H$ in the band storage form.

n — `INTEGER`. The order of matrix A ($n \geq 0$).

kd — `INTEGER`. The number of super-diagonals or sub-diagonals in the matrix A ($kd \geq 0$).

nrhs — `INTEGER`. The number of right-hand sides ($nrhs \geq 0$).

ab, b — `REAL` for `spbtrs`
`DOUBLE PRECISION` for `dpbtrs`
`COMPLEX` for `cpbtrs`
`DOUBLE COMPLEX` for `zpbtrs`.
Arrays: *ab*(*ldab*,*), *b*(*ldb*,*).

The array *ab* contains the Cholesky factor, as returned by the factorization routine, in *band storage* form.

The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations.

The second dimension of *ab* must be at least max(1, *n*),
the second dimension of *b* at least max(1,*nrhs*).

ldab — `INTEGER`. The first dimension of the array *ab*.
($ldab \geq kd + 1$).

ldb INTEGER. The first dimension of *b*; $ldb \ge \max(1, n)$.

Output Parameters

b Overwritten by the solution matrix *X*.

info INTEGER. If *info*=0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \le c(kd+1)\varepsilon P|U^H||U| \text{ or } |E| \le c(kd+1)\varepsilon P|L^H||L|$$

$c(k)$ is a modest linear function of k, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(kd+1)\,\mathrm{cond}(A, x)\varepsilon$$

where $\mathrm{cond}(A,x) = \|\,|A^{-1}|\,|A|\,|x|\,\|_\infty / \|x\|_\infty \le \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\mathrm{cond}(A,x)$ can be much smaller than $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector is $4n*kd$ for real flavors and $16n*kd$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?pbcon.
To refine the solution and estimate the error, call ?pbrfs.

?pttrs

Solves a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal matrix using the factorization computed by ?pttrf.

Syntax

```
call spttrs (n, nrhs, d, e, b, ldb, info)
call dpttrs (n, nrhs, d, e, b, ldb, info)
call cpttrs (uplo, n, nrhs, d, e, b, ldb, info)
```

```
call zpttrs (uplo, n, nrhs, d, e, b, ldb, info)
```

Description

This routine solves for X a system of linear equations $AX = B$ with a symmetric (Hermitian) positive-definite tridiagonal matrix A.
Before calling this routine, you must call ?pttrf to compute the LDL^H or U^HDU factorization of A.

Input Parameters

uplo	`CHARACTER*1`. Used for `cpttrs/zpttrs` only. Must be `'U'` or `'L'`. Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix A is stored and how A is factored: If *uplo* = `'U'`, the array *e* stores the superdiagonal of A, and A is factored as U^HDU; If *uplo* = `'L'`, the array *e* stores the subdiagonal of A, and A is factored as LDL^H.
n	`INTEGER`. The order of A ($n \ge 0$).
nrhs	`INTEGER`. The number of right-hand sides, i.e., the number of columns of the matrix B ($nrhs \ge 0$).
d	`REAL` for `spttrs, cpttrs` `DOUBLE PRECISION` for `dpttrs, zpttrs`. Array, dimension (*n*). Contains the diagonal elements of the diagonal matrix D from the factorization computed by ?pttrf.
e, b	`REAL` for `spttrs` `DOUBLE PRECISION` for `dpttrs` `COMPLEX` for `cpttrs` `DOUBLE COMPLEX` for `zpttrs`. Arrays: *e*(*n* - 1), *b*(*ldb*,*nrhs*). The array *e* contains the (n - 1) off-diagonal elements of the unit bidiagonal factor U or L from the factorization computed by ?pttrf (see *uplo*). The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations.
ldb	`INTEGER`. The leading dimension of *b*; $ldb \ge \max(1, n)$.

Output Parameters

b	Overwritten by the solution matrix X.

info INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

?sytrs

Solves a system of linear equations with a UDU- or LDL-factored symmetric matrix.

Syntax

```
call ssytrs (uplo, n, nrhs, a, lda, ipiv, b, ldb, info)
call dsytrs (uplo, n, nrhs, a, lda, ipiv, b, ldb, info)
call csytrs (uplo, n, nrhs, a, lda, ipiv, b, ldb, info)
call zsytrs (uplo, n, nrhs, a, lda, ipiv, b, ldb, info)
```

Description

This routine solves for X the system of linear equations $AX = B$ with a symmetric matrix A, given the Bunch-Kaufman factorization of A:

if *uplo*='U', $A = PUDU^T P^T$

if *uplo*='L', $A = PLDL^T P^T$

where P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B. You must supply to this routine the factor U (or L) and the array *ipiv* returned by the factorization routine ?sytrf.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix A has been factored:
If *uplo* = 'U', the array *a* stores the upper triangular factor U of the factorization $A = PUDU^T P^T$.
If *uplo* = 'L', the array *a* stores the lower triangular factor L of the factorization $A = PLDL^T P^T$.

n INTEGER. The order of matrix A ($n \geq 0$).

nrhs INTEGER. The number of right-hand sides ($nrhs \geq 0$).

ipiv — `INTEGER`. Array, `DIMENSION` at least max(1,*n*). The *ipiv* array, as returned by `?sytrf`.

a, b — `REAL` for `ssytrs`
`DOUBLE PRECISION` for `dsytrs`
`COMPLEX` for `csytrs`
`DOUBLE COMPLEX` for `zsytrs`.
Arrays: `a(lda,*)`, `b(ldb,*)`.
The array *a* contains the factor *U* or *L* (see *uplo*).
The array *b* contains the matrix *B* whose columns are the right-hand sides for the system of equations.

The second dimension of *a* must be at least max(1,*n*), the second dimension of *b* at least max(1,*nrhs*).

lda — `INTEGER`. The first dimension of *a*; $lda \geq \max(1, n)$.

ldb — `INTEGER`. The first dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

b — Overwritten by the solution matrix *X*.

info — `INTEGER`. If *info*=0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon P|U||D||U^T|P^T \quad \text{or} \quad |E| \leq c(n)\varepsilon P|L||D||L^T|P^T$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n)\,\mathrm{cond}(A, x)\varepsilon$$

where $\mathrm{cond}(A,x) = \|\,|A^{-1}|\,|A|\,|x|\,\|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\mathrm{cond}(A,x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $2n^2$ for real flavors or $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?sycon.
To refine the solution and estimate the error, call ?syrfs.

?hetrs

Solves a system of linear equations with a UDU- or LDL-factored Hermitian matrix.

Syntax

```
call chetrs (uplo, n, nrhs, a, lda, ipiv, b, ldb, info)
call zhetrs (uplo, n, nrhs, a, lda, ipiv, b, ldb, info)
```

Description

This routine solves for X the system of linear equations $AX = B$ with a Hermitian matrix A, given the Bunch-Kaufman factorization of A:

if `uplo` = 'U', $A = PUDU^HP^T$

if `uplo` = 'L', $A = PLDL^HP^T$

where P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B. You must supply to this routine the factor U (or L) and the array *ipiv* returned by the factorization routine ?hetrf.

Input Parameters

uplo — `CHARACTER*1`. Must be 'U' or 'L'.
Indicates how the input matrix A has been factored:

If *uplo* = 'U', the array *a* stores the upper triangular factor U of the factorization $A = PUDU^HP^T$.

If *uplo* = 'L', the array *a* stores the lower triangular factor L of the factorization $A = PLDL^HP^T$.

n — `INTEGER`. The order of matrix A ($n \ge 0$).

nrhs — `INTEGER`. The number of right-hand sides ($nrhs \ge 0$).

ipiv	INTEGER. Array, DIMENSION at least max(1,*n*). The *ipiv* array, as returned by ?hetrf.
a, b	COMPLEX for chetrs. DOUBLE COMPLEX for zhetrs. Arrays: *a*(*lda*,*), *b*(*ldb*,*). The array *a* contains the factor *U* or *L* (see *uplo*). The array *b* contains the matrix *B* whose columns are the right-hand sides for the system of equations. The second dimension of *a* must be at least max(1,*n*), the second dimension of *b* at least max(1,*nrhs*).
lda	INTEGER. The first dimension of *a*; $lda \geq \max(1, n)$.
ldb	INTEGER. The first dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon P|U||D||U^H|P^T \text{ or } |E| \leq c(n)\varepsilon P|L||D||L^H|P^T$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n)\,\mathrm{cond}(A, x)\varepsilon$$

where $\mathrm{cond}(A,x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\mathrm{cond}(A,x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $8n^2$.

To estimate the condition number $\kappa_\infty(A)$, call ?hecon.
To refine the solution and estimate the error, call ?herfs.

?sptrs

Solves a system of linear equations with a UDU- or LDL-factored symmetric matrix using packed storage.

Syntax

```
call ssptrs ( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call dsptrs ( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call csptrs ( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zsptrs ( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

Description

This routine solves for X the system of linear equations $AX = B$ with a symmetric matrix A, given the Bunch-Kaufman factorization of A:

if `uplo`=`'U'`, $A = PUDU^TP^T$
if `uplo`=`'L'`, $A = PLDL^TP^T$

where P is a permutation matrix, U and L are upper and lower ***packed*** triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B. You must supply the factor U (or L) and the array `ipiv` returned by the factorization routine `?sptrf`.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates how the input matrix A has been factored:
If `uplo` = `'U'`, the array `ap` stores the packed factor U of the factorization $A = PUDU^TP^T$.
If `uplo` = `'L'`, the array `ap` stores the packed factor L of the factorization $A = PLDL^TP^T$.

n — `INTEGER`. The order of matrix A ($n \geq 0$).

nrhs — `INTEGER`. The number of right-hand sides ($nrhs \geq 0$).

ipiv — `INTEGER`. Array, `DIMENSION` at least max(1,n).
The `ipiv` array, as returned by `?sptrf`.

ap, b — `REAL` for `ssptrs`
`DOUBLE PRECISION` for `dsptrs`
`COMPLEX` for `csptrs`

DOUBLE COMPLEX for `zsptrs`.
Arrays: *ap*(*), *b*(*ldb*,*)
The dimension of *ap* must be at least max(1,$n(n+1)/2$).
The array *ap* contains the factor *U* or *L*, as specified by *uplo*, in *packed storage* (see Matrix Storage Schemes).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the system of equations. The second dimension of *b* must be at least max(1,*nrhs*).

ldb — INTEGER. The first dimension of *b*; $ldb \ge \max(1, n)$.

Output Parameters

b — Overwritten by the solution matrix *X*.

info — INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \le c(n)\varepsilon P|U||D||U^T|P^T \quad \text{or} \quad |E| \le c(n)\varepsilon P|L||D||L^T|P^T$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(n)\ \mathrm{cond}(A, x)\varepsilon$$

where $\mathrm{cond}(A,x) = \|\ |A^{-1}|\ |A|\ |x|\ \|_\infty / \|x\|_\infty \le \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that cond(A,x) can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $2n^2$ for real flavors or $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call `?spcon`.
To refine the solution and estimate the error, call `?sprfs`.

?hptrs

Solves a system of linear equations with a UDU- or LDL-factored Hermitian matrix using packed storage.

Syntax

```
call chptrs ( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zhptrs ( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

Description

This routine solves for X the system of linear equations $AX = B$ with a Hermitian matrix A, given the Bunch-Kaufman factorization of A:

if `uplo`=`'U'`, $A = PUDU^H P^T$
if `uplo`=`'L'`, $A = PLDL^H P^T$

where P is a permutation matrix, U and L are upper and lower ***packed*** triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B.

You must supply to this routine the arrays `ap` (containing U or L) and `ipiv` in the form returned by the factorization routine `?hptrf`.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Indicates how the input matrix A has been factored: If *uplo* = `'U'`, the array *ap* stores the packed factor U of the factorization $A = PUDU^H P^T$. If *uplo* = `'L'`, the array *ap* stores the packed factor L of the factorization $A = PLDL^H P^T$.
n	`INTEGER`. The order of matrix A ($n \ge 0$).
nrhs	`INTEGER`. The number of right-hand sides ($nrhs \ge 0$).
ipiv	`INTEGER`. Array, `DIMENSION` at least max(1,*n*). The *ipiv* array, as returned by `?hptrf`.
ap, b	`COMPLEX` for `chptrs`. `DOUBLE COMPLEX` for `zhptrs`. Arrays: *ap*(*), *b*(*ldb*,*)

The dimension of *ap* must be at least max(1,$n(n+1)/2$).
The array *ap* contains the factor U or L, as specified by *uplo*, in *packed storage* (see Matrix Storage Schemes).

The array *b* contains the matrix B whose columns are the right-hand sides for the system of equations. The second dimension of *b* must be at least max(1,*nrhs*).

ldb INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

b Overwritten by the solution matrix X.

info INTEGER. If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

For each right-hand side b, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \le c(n)\varepsilon P|U||D||U^H|P^T \text{ or } |E| \le c(n)\varepsilon P|L||D||L^H|P^T$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(n)\,\mathrm{cond}(A, x)\varepsilon$$

where $\mathrm{cond}(A,x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \le \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\mathrm{cond}(A,x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?hpcon.
To refine the solution and estimate the error, call ?hprfs.

?trtrs

Solves a system of linear equations with a triangular matrix, with multiple right-hand sides.

Syntax

```
call strtrs (uplo,trans,diag,n,nrhs,a, lda,b, ldb,info)
call dtrtrs (uplo,trans,diag,n,nrhs,a, lda,b, ldb,info)
call ctrtrs (uplo,trans,diag,n,nrhs,a, lda,b, ldb,info)
call ztrtrs (uplo,trans,diag,n,nrhs,a, lda,b, ldb,info)
```

Description

This routine solves for X the following systems of linear equations with a triangular matrix A, with multiple right-hand sides stored in B:

$AX = B$	if *trans*='N',
$A^T X = B$	if *trans*='T',
$A^H X = B$	if *trans*='C' (for complex matrices only).

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.

Indicates whether A is upper or lower triangular:

If *uplo* = 'U', then A is upper triangular.
If *uplo* = 'L', then A is lower triangular.

trans CHARACTER*1. Must be 'N' or 'T' or 'C'.
If *trans* = 'N', then $AX = B$ is solved for X.
If *trans* = 'T', then $A^T X = B$ is solved for X.
If *trans* = 'C', then $A^H X = B$ is solved for X.

diag CHARACTER*1. Must be 'N' or 'U'.

If *diag* = 'N', then A is not a unit triangular matrix.
If *diag* = 'U', then A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array *a*.

n INTEGER. The order of A; the number of rows in B ($n \geq 0$).

nrhs INTEGER. The number of right-hand sides ($nrhs \geq 0$).

a, b — `REAL` for `strtrs`
`DOUBLE PRECISION` for `dtrtrs`
`COMPLEX` for `ctrtrs`
`DOUBLE COMPLEX` for `ztrtrs`.
Arrays: *a*(*lda*,*), *b*(*ldb*,*).

The array *a* contains the matrix *A*.
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

The second dimension of *a* must be at least max(1,*n*), the second dimension of *b* at least max(1,*nrhs*).

lda — `INTEGER`. The first dimension of *a*; *lda* ≥ max(1, *n*).

ldb — `INTEGER`. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

b — Overwritten by the solution matrix *X*.

info — `INTEGER`. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$ where

$$|E| \le c(n)\varepsilon\, |A|$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.
If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(n)\ \mathrm{cond}(A, x)\varepsilon, \quad \text{provided } c(n)\ \mathrm{cond}(A, x)\varepsilon < 1$$

where $\mathrm{cond}(A,x) = \|\, |A^{-1}|\, |A|\, |x|\, \|_\infty / \|x\|_\infty \le \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that cond(A,x) can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is n^2 for real flavors and $4n^2$ for complex flavors.

To estimate the condition number $\kappa_{\infty}(A)$, call ?trcon.
To estimate the error in the solution, call ?trrfs.

?tptrs

Solves a system of linear equations with a packed triangular matrix, with multiple right-hand sides.

Syntax

```
call stptrs (uplo, trans, diag, n, nrhs, ap, b, ldb, info)
call dtptrs (uplo, trans, diag, n, nrhs, ap, b, ldb, info)
call ctptrs (uplo, trans, diag, n, nrhs, ap, b, ldb, info)
call ztptrs (uplo, trans, diag, n, nrhs, ap, b, ldb, info)
```

Description

This routine solves for X the following systems of linear equations with a packed triangular matrix A, with multiple right-hand sides stored in B:

$AX = B$	if `trans='N'`,
$A^TX = B$	if `trans='T'`,
$A^HX = B$	if `trans='C'` (for complex matrices only).

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates whether A is upper or lower triangular:

If *uplo* = 'U', then A is upper triangular.
If *uplo* = 'L', then A is lower triangular.

trans CHARACTER*1. Must be 'N' or 'T' or 'C'.
If *trans* = 'N', then $AX = B$ is solved for X.
If *trans* = 'T', then $A^TX = B$ is solved for X.
If *trans* = 'C', then $A^HX = B$ is solved for X.

diag CHARACTER*1. Must be 'N' or 'U'.

If *diag* = 'N', then A is not a unit triangular matrix.

If *diag* = 'U', then A is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array *ap*.

n	`INTEGER`. The order of A; the number of rows in B ($n \ge 0$).
nrhs	`INTEGER`. The number of right-hand sides ($nrhs \ge 0$).
ap, b	`REAL` for `stptrs` `DOUBLE PRECISION` for `dtptrs` `COMPLEX` for `ctptrs` `DOUBLE COMPLEX` for `ztptrs`. Arrays: `ap(*)`, `b(ldb,*)` The dimension of `ap` must be at least max(1,*n*(*n*+1)/2). The array `ap` contains the matrix A in *packed storage* (see Matrix Storage Schemes). The array `b` contains the matrix B whose columns are the right-hand sides for the system of equations. The second dimension of `b` must be at least max(1,*nrhs*).
ldb	`INTEGER`. The first dimension of *b*; $ldb \ge \max(1, n)$.

Output Parameters

b	Overwritten by the solution matrix X.
info	`INTEGER`. If *info*=0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For each right-hand side b, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$ where

$$|E| \le c(n)\varepsilon\,|A|$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(n)\ \mathrm{cond}(A, x)\varepsilon, \text{ provided } c(n)\ \mathrm{cond}(A, x)\varepsilon < 1$$

where $\mathrm{cond}(A, x) = \|\,|A^{-1}|\,|A|\,|x|\,\|_\infty / \|x\|_\infty \le \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\mathrm{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector *b* is n^2 for real flavors and $4n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call `?tpcon`.
To estimate the error in the solution, call `?tprfs`.

?tbtrs

Solves a system of linear equations with a band triangular matrix, with multiple right-hand sides.

Syntax

```
call stbtrs (uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info)
call dtbtrs (uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info)
call ctbtrs (uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info)
call ztbtrs (uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info)
```

Description

This routine solves for *X* the following systems of linear equations with a band triangular matrix *A*, with multiple right-hand sides stored in *B*:

$AX = B$	if *trans*='N',
$A^TX = B$	if *trans*='T',
$A^HX = B$	if *trans*='C' (for complex matrices only).

Input Parameters

uplo — `CHARACTER*1`. Must be 'U' or 'L'.
Indicates whether *A* is upper or lower triangular:

If *uplo* = 'U', then *A* is upper triangular.
If *uplo* = 'L', then *A* is lower triangular.

trans — `CHARACTER*1`. Must be 'N' or 'T' or 'C'.
If *trans* = 'N', then $AX = B$ is solved for *X*.
If *trans* = 'T', then $A^TX = B$ is solved for *X*.
If *trans* = 'C', then $A^HX = B$ is solved for *X*.

diag — `CHARACTER*1`. Must be 'N' or 'U'.
If *diag* = 'N', then *A* is not a unit triangular matrix.

If *diag* = 'U', then A is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array *ab*.

n — INTEGER. The order of A; the number of rows in B ($n \geq 0$).

kd — INTEGER. The number of super-diagonals or sub-diagonals in the matrix A ($kd \geq 0$).

nrhs — INTEGER. The number of right-hand sides ($nrhs \geq 0$).

ab, b — REAL for stbtrs
DOUBLE PRECISION for dtbtrs
COMPLEX for ctbtrs
DOUBLE COMPLEX for ztbtrs.
Arrays: *ab*(*ldab*,*), *b*(*ldb*,*).

The array *ab* contains the matrix A in *band storage* form.

The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations.

The second dimension of *ab* must be at least max(1, *n*),
the second dimension of *b* at least max(1,*nrhs*).

ldab — INTEGER. The first dimension of *ab*; $ldab \geq kd + 1$.

ldb — INTEGER. The first dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

b — Overwritten by the solution matrix X.

info — INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For each right-hand side b, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$ where

$$|E| \leq c(n)\varepsilon |A|$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.
If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n)\,\mathrm{cond}(A, x)\varepsilon, \text{ provided } c(n)\,\mathrm{cond}(A, x)\varepsilon < 1$$

where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \le \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that cond(A, x) can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is 2`n*kd` for real flavors and 8`n*kd` for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call `?tbcon`.
To estimate the error in the solution, call `?tbrfs`.

Routines for Estimating the Condition Number

This section describes the LAPACK routines for estimating the *condition number* of a matrix. The condition number is used for analyzing the errors in the solution of a system of linear equations (see Error Analysis). Since the condition number may be arbitrarily large when the matrix is nearly singular, the routines actually compute the *reciprocal* condition number.

?gecon

Estimates the reciprocal of the condition number of a general matrix in either the 1-norm or the infinity-norm.

Syntax

```
call sgecon ( norm, n, a, lda, anorm, rcond, work, iwork, info )
call dgecon ( norm, n, a, lda, anorm, rcond, work, iwork, info )
call cgecon ( norm, n, a, lda, anorm, rcond, work, rwork, info )
call zgecon ( norm, n, a, lda, anorm, rcond, work, rwork, info )
```

Description

This routine estimates the reciprocal of the condition number of a general matrix A in either the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$
$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).$$

Before calling this routine:

- compute `anorm` (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call `?getrf` to compute the *LU* factorization of *A*.

Input Parameters

norm	`CHARACTER*1`. Must be `'1'` or `'O'` or `'I'`. If *norm* = `'1'` or `'O'`, then the routine estimates $\kappa_1(A)$. If *norm* = `'I'`, then the routine estimates $\kappa_\infty(A)$.
n	`INTEGER`. The order of the matrix *A* ($n \geq 0$).
a, work	`REAL` for `sgecon` `DOUBLE PRECISION` for `dgecon` `COMPLEX` for `cgecon` `DOUBLE COMPLEX` for `zgecon`. Arrays: `a(lda,*)`, `work(*)`. The array `a` contains the *LU*-factored matrix *A*, as returned by `?getrf`. The second dimension of `a` must be at least max(1,*n*). The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 4**n*) for real flavors and max(1, 2**n*) for complex flavors.
anorm	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. The norm of the *original* matrix *A* (see Description).
lda	`INTEGER`. The first dimension of `a`; *lda* ≥ max(1, *n*).
iwork	`INTEGER`. Workspace array, `DIMENSION` at least max(1, *n*).
rwork	`REAL` for `cgecon` `DOUBLE PRECISION` for `zgecon` Workspace array, `DIMENSION` at least max(1, 2**n*).

Output Parameters

rcond	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The computed *rcond* is never less than ρ (the reciprocal of the true condition number) and in practice is nearly always less than 10ρ. A call to this routine involves solving a number of systems of linear equations $Ax = b$ or $A^H x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?gbcon

Estimates the reciprocal of the condition number of a band matrix in either the 1-norm or the infinity-norm.

Syntax

```
call sgbcon (norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, iwork, info)
call dgbcon (norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, iwork, info)
call cgbcon (norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, rwork, info)
call zgbcon (norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, rwork, info)
```

Description

This routine estimates the reciprocal of the condition number of a general band matrix A in either the 1-norm or infinity-norm:

$$\kappa_1(A) = ||A||_1 \, ||A^{-1}||_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$
$$\kappa_\infty(A) = ||A||_\infty \, ||A^{-1}||_\infty = \kappa_1(A^T) = \kappa_1(A^H) .$$

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?gbtrf to compute the *LU* factorization of *A*.

Input Parameters

norm CHARACTER*1. Must be '1' or 'O' or 'I'.

If *norm* = '1' or 'O', then the routine estimates $\kappa_1(A)$.
If *norm* = 'I', then the routine estimates $\kappa_\infty(A)$.

n	`INTEGER`. The order of the matrix *A* ($n \geq 0$).
kl	`INTEGER`. The number of sub-diagonals within the band of *A* ($kl \geq 0$).
ku	`INTEGER`. The number of super-diagonals within the band of *A* ($ku \geq 0$).
ldab	`INTEGER`. The first dimension of the array *ab*. ($ldab \geq 2kl + ku + 1$).
ipiv	`INTEGER`. Array, `DIMENSION` at least max(1,*n*). The *ipiv* array, as returned by `?gbtrf`.
ab, work	`REAL` for `sgbcon` `DOUBLE PRECISION` for `dgbcon` `COMPLEX` for `cgbcon` `DOUBLE COMPLEX` for `zgbcon`. Arrays: *ab*(*ldab*,*), *work*(*). The array *ab* contains the factored band matrix *A*, as returned by `?gbtrf`. The second dimension of *ab* must be at least max(1,*n*). The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.
anorm	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. The norm of the *original* matrix *A* (see Description).
iwork	`INTEGER`. Workspace array, `DIMENSION` at least max(1, *n*).
rwork	`REAL` for `cgbcon` `DOUBLE PRECISION` for `zgbcon` Workspace array, `DIMENSION` at least max(1, 2*n).

Output Parameters

rcond	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info `INTEGER`. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The computed *rcond* is never less than ρ (the reciprocal of the true condition number) and in practice is nearly always less than 10ρ. A call to this routine involves solving a number of systems of linear equations $Ax = b$ or $A^H x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n(ku + 2kl)$ floating-point operations for real flavors and $8n(ku + 2kl)$ for complex flavors.

?gtcon

Estimates the reciprocal of the condition number of a tridiagonal matrix using the factorization computed by `?gttrf`.

Syntax

```
call sgtcon ( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, iwork, info )
call dgtcon ( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, iwork, info )
call cgtcon ( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, info )
call zgtcon ( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, info )
```

Description

This routine estimates the reciprocal of the condition number of a real or complex tridiagonal matrix A in either the 1-norm or infinity-norm:

$$\kappa_1(A) = ||A||_1 \; ||A^{-1}||_1$$
$$\kappa_\infty(A) = ||A||_\infty \; ||A^{-1}||_\infty$$

An estimate is obtained for $||A^{-1}||$, and the reciprocal of the condition number is computed as $rcond = 1 / (||A|| \; ||A^{-1}||)$.

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call `?gttrf` to compute the LU factorization of A.

Input Parameters

norm — `CHARACTER*1`. Must be `'1'` or `'O'` or `'I'`.
If *norm* = `'1'` or `'O'`, then the routine estimates $\kappa_1(A)$.
If *norm* = `'I'`, then the routine estimates $\kappa_\infty(A)$.

n — `INTEGER`. The order of the matrix A ($n \geq 0$).

dl,d,du,du2 — `REAL` for `sgtcon`
`DOUBLE PRECISION` for `dgtcon`
`COMPLEX` for `cgtcon`
`DOUBLE COMPLEX` for `zgtcon`.
Arrays: *dl*(*n* - 1), *d*(*n*), *du*(*n* - 1), *du2*(*n* - 2).
The array *dl* contains the (n - 1) multipliers that define the matrix L from the LU factorization of A as computed by `?gttrf`.
The array *d* contains the *n* diagonal elements of the upper triangular matrix U from the LU factorization of A.
The array *du* contains the (n - 1) elements of the first super-diagonal of U.
The array *du2* contains the (n - 2) elements of the second super-diagonal of U.

ipiv — `INTEGER`.
Array, `DIMENSION (`*n*`)`.
The array of pivot indices, as returned by `?gttrf`.

anorm — `REAL` for single precision flavors.
`DOUBLE PRECISION` for double precision flavors.
The norm of the *original* matrix A (see *Description*).

work — `REAL` for `sgtcon`
`DOUBLE PRECISION` for `dgtcon`
`COMPLEX` for `cgtcon`
`DOUBLE COMPLEX` for `zgtcon`.
Workspace array, `DIMENSION (2*`*n*`)`.

iwork — `INTEGER`.
Workspace array, `DIMENSION (`*n*`)`.
Used for real flavors only.

Output Parameters

rcond — `REAL` for single precision flavors.
`DOUBLE PRECISION` for double precision flavors.
An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working

precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The computed *rcond* is never less than ρ (the reciprocal of the true condition number) and in practice is nearly always less than 10ρ. A call to this routine involves solving a number of systems of linear equations $Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?pocon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite matrix.

Syntax

```
call spocon ( uplo, n, a, lda, anorm, rcond, work, iwork, info )
call dpocon ( uplo, n, a, lda, anorm, rcond, work, iwork, info )
call cpocon ( uplo, n, a, lda, anorm, rcond, work, rwork, info )
call zpocon ( uplo, n, a, lda, anorm, rcond, work, rwork, info )
```

Description

This routine estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite matrix A:

$\kappa_1(A) = ||A||_1 \, ||A^{-1}||_1$ (since A is symmetric or Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?potrf to compute the Cholesky factorization of A.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Indicates how the input matrix A has been factored: If *uplo* = `'U'`, the array *a* stores the upper triangular factor U of the factorization $A = U^H U$. If *uplo* = `'L'`, the array *a* stores the lower triangular factor L of the factorization $A = LL^H$.
n	`INTEGER`. The order of the matrix A ($n \geq 0$).
a, work	`REAL` for `spocon` `DOUBLE PRECISION` for `dpocon` `COMPLEX` for `cpocon` `DOUBLE COMPLEX` for `zpocon`. Arrays: `a(lda,*)`, `work(*)`. The array *a* contains the factored matrix A, as returned by `?potrf`. The second dimension of *a* must be at least max(1,*n*). The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.
lda	`INTEGER`. The first dimension of *a*; *lda* ≥ max(1, *n*).
anorm	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. The norm of the *original* matrix A (see *Description*).
iwork	`INTEGER`. Workspace array, `DIMENSION` at least max(1, *n*).
rwork	`REAL` for `cpocon` `DOUBLE PRECISION` for `zpocon` Workspace array, `DIMENSION` at least max(1, *n*).

Output Parameters

rcond	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working

precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The computed *rcond* is never less than ρ (the reciprocal of the true condition number) and in practice is nearly always less than 10ρ. A call to this routine involves solving a number of systems of linear equations $Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?ppcon

Estimates the reciprocal of the condition number of a packed symmetric (Hermitian) positive-definite matrix.

Syntax

```
call sppcon ( uplo, n, ap, anorm, rcond, work, iwork, info )
call dppcon ( uplo, n, ap, anorm, rcond, work, iwork, info )
call cppcon ( uplo, n, ap, anorm, rcond, work, rwork, info )
call zppcon ( uplo, n, ap, anorm, rcond, work, rwork, info )
```

Description

This routine estimates the reciprocal of the condition number of a packed symmetric (Hermitian) positive-definite matrix A:

$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1$ (since A is symmetric or Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \Sigma_i |a_{ij}|$ or $\|A\|_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?pptrf to compute the Cholesky factorization of A.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates how the input matrix A has been factored:

If *uplo* = `'U'`, the array *ap* stores the upper triangular factor U of the factorization $A = U^H U$.

If *uplo* = `'L'`, the array *ap* stores the lower triangular factor L of the factorization $A = LL^H$.

n — `INTEGER`. The order of the matrix A ($n \ge 0$).

ap, work — `REAL` for `sppcon`
`DOUBLE PRECISION` for `dppcon`
`COMPLEX` for `cppcon`
`DOUBLE COMPLEX` for `zppcon`.
Arrays: *ap*(*), *work*(*).

The array *ap* contains the packed factored matrix A, as returned by `?pptrf`.
The dimension of *ap* must be at least max(1,n(n+1)/2).
The array *work* is a workspace for the routine.

The dimension of *work* must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.

anorm — `REAL` for single precision flavors.
`DOUBLE PRECISION` for double precision flavors.
The norm of the *original* matrix A (see *Description*).

iwork — `INTEGER`.
Workspace array, `DIMENSION` at least max(1, n).

rwork — `REAL` for `cppcon`
`DOUBLE PRECISION` for `zppcon`
Workspace array, `DIMENSION` at least max(1, n).

Output Parameters

rcond — `REAL` for single precision flavors.
`DOUBLE PRECISION` for double precision flavors.
An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The computed *rcond* is never less than ρ (the reciprocal of the true condition number) and in practice is nearly always less than 10ρ. A call
to this routine involves solving a number of systems of linear equations
$Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?pbcon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite band matrix.

Syntax

```
call spbcon (uplo, n, kd, ab, ldab, anorm, rcond, work, iwork, info)
call dpbcon (uplo, n, kd, ab, ldab, anorm, rcond, work, iwork, info)
call cpbcon (uplo, n, kd, ab, ldab, anorm, rcond, work, rwork, info)
call zpbcon (uplo, n, kd, ab, ldab, anorm, rcond, work, rwork, info)
```

Description

This routine estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite band matrix A:

$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1$ (since A is symmetric or Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \Sigma_i |a_{ij}|$ or $\|A\|_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?pbtrf to compute the Cholesky factorization of A.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix A has been factored:
If *uplo* = 'U', the array *ab* stores the upper triangular factor U of the

	Cholesky factorization $A = U^H U$. If `uplo` = `'L'`, the array `ab` stores the lower triangular factor L of the factorization $A = LL^H$.
n	`INTEGER`. The order of the matrix A ($n \geq 0$).
kd	`INTEGER`. The number of super-diagonals or sub-diagonals in the matrix A ($kd \geq 0$).
ldab	`INTEGER`. The first dimension of the array `ab`. ($ldab \geq kd + 1$).
ab, work	`REAL` for `spbcon` `DOUBLE PRECISION` for `dpbcon` `COMPLEX` for `cpbcon` `DOUBLE COMPLEX` for `zpbcon`. Arrays: `ab(ldab,*)`, `work(*)`. The array `ab` contains the factored matrix A in band form, as returned by `?pbtrf`. The second dimension of `ab` must be at least max(1, *n*), The array `work` is a workspace for the routine. The dimension of `work` must be at least max(1, 3*n*) for real flavors and max(1, 2*n*) for complex flavors.
anorm	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. The norm of the *original* matrix A (see *Description*).
iwork	`INTEGER`. Workspace array, `DIMENSION` at least max(1, *n*).
rwork	`REAL` for `cpbcon` `DOUBLE PRECISION` for `zpbcon`. Workspace array, `DIMENSION` at least max(1, *n*).

Output Parameters

rcond	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets `rcond` =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime `rcond` is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

`info` INTEGER. If `info`=0, the execution is successful.
If `info` = `-i`, the `i`th parameter had an illegal value.

Application Notes

The computed `rcond` is never less than ρ (the reciprocal of the true condition number) and in practice is nearly always less than 10ρ. A call
to this routine involves solving a number of systems of linear equations
$Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $4n(kd + 1)$ floating-point operations for real flavors and $16n(kd + 1)$ for complex flavors.

?ptcon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite tridiagonal matrix.

Syntax

```
call sptcon (n, d, e, anorm, rcond, work, info)
call dptcon (n, d, e, anorm, rcond, work, info)
call cptcon (n, d, e, anorm, rcond, work, info)
call zptcon (n, d, e, anorm, rcond, work, info)
```

Description

This routine computes the reciprocal of the condition number (in the 1-norm) of a real symmetric or complex Hermitian positive-definite tridiagonal matrix using the factorization $A = LDL^H$ or $A = U^H DU$ computed by ?pttrf :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric or Hermitian, } \kappa_\infty(A) = \kappa_1(A)).$$

The norm $\|A^{-1}\|$ is computed by a direct method, and the reciprocal of the condition number is computed as `rcond` $= 1 / (\|A\| \|A^{-1}\|)$.

Before calling this routine:

- compute `anorm` as $\|A\|_1 = \max_j \Sigma_i |a_{ij}|$
- call ?pttrf to compute the factorization of A.

Input Parameters

n	INTEGER. The order of the matrix *A* ($n \geq 0$).
d, work	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, dimension (*n*). The array *d* contains the *n* diagonal elements of the diagonal matrix *D* from the factorization of *A,* as computed by ?pttrf ; *work* is a workspace array.
e	REAL for sptcon DOUBLE PRECISION for dptcon COMPLEX for cptcon DOUBLE COMPLEX for zptcon. Array, DIMENSION (n - 1). Contains off-diagonal elements of the unit bidiagonal factor *U* or *L* from the factorization computed by ?pttrf .
anorm	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The 1- norm of the *original* matrix *A* (see *Description*).

Output Parameters

rcond	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The computed *rcond* is never less than ρ (the reciprocal of the true condition number) and in practice is nearly always less than 10ρ. A call
to this routine involves solving a number of systems of linear equations
$Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $4n(kd + 1)$ floating-point operations for real flavors and $16n(kd + 1)$ for complex flavors.

?sycon

Estimates the reciprocal of the condition number of a symmetric matrix.

Syntax

```
call ssycon (uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info)
call dsycon (uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info)
call csycon (uplo, n, a, lda, ipiv, anorm, rcond, work, rwork, info)
call zsycon (uplo, n, a, lda, ipiv, anorm, rcond, work, rwork, info)
```

Description

This routine estimates the reciprocal of the condition number of a symmetric matrix A:

$\kappa_1(A) = \|A\|_1 \, \|A^{-1}\|_1$ (since A is symmetric, $\kappa_\infty(A) = \kappa_1(A)$).

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \Sigma_i |a_{ij}|$ or $\|A\|_\infty = \max_i \Sigma_j |a_{ij}|$)
- call `?sytrf` to compute the factorization of A.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates how the input matrix A has been factored:
If *uplo* = `'U'`, the array *a* stores the upper triangular factor U of the factorization $A = PUDU^TP^T$.
If *uplo* = `'L'`, the array *a* stores the lower triangular factor L of the factorization $A = PLDL^TP^T$.

n — `INTEGER`. The order of matrix A ($n \ge 0$).

a, work — `REAL` for `ssycon`
`DOUBLE PRECISION` for `dsycon`
`COMPLEX` for `csycon`
`DOUBLE COMPLEX` for `zsycon`.
Arrays: *a*(*lda*,*), *work*(*).

The array *a* contains the factored matrix A, as returned by `?sytrf`. The second dimension of *a* must be at least max(1,*n*).

The array *work* is a workspace for the routine.

	The dimension of `work` must be at least max(1, 2*n).
lda	`INTEGER`. The first dimension of *a*; *lda* ≥ max(1, *n*).
ipiv	`INTEGER`. Array, `DIMENSION` at least max(1,*n*). The array *ipiv*, as returned by `?sytrf`.
anorm	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. The norm of the *original* matrix *A* (see *Description*).
iwork	`INTEGER`. Workspace array, `DIMENSION` at least max(1, *n*).
rwork	`REAL` for `csycon` `DOUBLE PRECISION` for `zsycon`. Workspace array, `DIMENSION` at least max(1, *n*).

Output Parameters

rcond	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

The computed *rcond* is never less than ρ (the reciprocal of the true condition number) and in practice is nearly always less than 10ρ. A call to this routine involves solving a number of systems of linear equations $Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?hecon

Estimates the reciprocal of the condition number of a Hermitian matrix.

Syntax

```
call checon (uplo, n, a, lda, ipiv, anorm, rcond, work, rwork, info)
call zhecon (uplo, n, a, lda, ipiv, anorm, rcond, work, rwork, info)
```

Description

This routine estimates the reciprocal of the condition number of a Hermitian matrix A:

$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1$ (since A is Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \Sigma_i |a_{ij}|$ or $\|A\|_\infty = \max_i \Sigma_j |a_{ij}|$)
- call `?hetrf` to compute the factorization of A.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates how the input matrix A has been factored:

If *uplo* = `'U'`, the array *a* stores the upper triangular factor U of the factorization $A = PUDU^HP^T$.

If *uplo* = `'L'`, the array *a* stores the lower triangular factor L of the factorization $A = PLDL^HP^T$.

n — `INTEGER`. The order of matrix A ($n \ge 0$).

a, work — `COMPLEX` for `checon`
`DOUBLE COMPLEX` for `zhecon`.
Arrays: *a*(*lda*,*), *work*(*).

The array *a* contains the factored matrix A, as returned by `?hetrf`.
The second dimension of *a* must be at least max(1,*n*).

The array *work* is a workspace for the routine.
The dimension of *work* must be at least max(1, 2**n*).

lda — `INTEGER`. The first dimension of *a*; *lda* ≥ max(1, *n*).

ipiv `INTEGER`. Array, `DIMENSION` at least max(1,*n*).
The array *ipiv*, as returned by ?hetrf.

anorm `REAL` for single precision flavors.
`DOUBLE PRECISION` for double precision flavors.
The norm of the *original* matrix *A* (see *Discussion*).

rwork `REAL` for `checon`
`DOUBLE PRECISION` for `zhecon`
Workspace array, `DIMENSION` at least max(1, *n*).

Output Parameters

rcond `REAL` for single precision flavors.
`DOUBLE PRECISION` for double precision flavors.
An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

The computed *rcond* is never less than ρ (the reciprocal of the true condition number) and in practice is nearly always less than 10ρ. A call to this routine involves solving a number of systems of linear equations $Ax = b$; the number is usually 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

?spcon

Estimates the reciprocal of the condition number of a packed symmetric matrix.

Syntax

```
call sspcon ( uplo, n, ap, ipiv, anorm, rcond, work, iwork, info )
```

```
call dspcon ( uplo, n, ap, ipiv, anorm, rcond, work, iwork, info )
call cspcon ( uplo, n, ap, ipiv, anorm, rcond, work, rwork, info )
call zspcon ( uplo, n, ap, ipiv, anorm, rcond, work, rwork, info )
```

Description

This routine estimates the reciprocal of the condition number of a packed symmetric matrix *A*:

$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1$ (since *A* is symmetric, $\kappa_\infty(A) = \kappa_1(A)$).

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \Sigma_i |a_{ij}|$ or $\|A\|_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?sptrf to compute the factorization of *A*.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix *A* has been factored:
If *uplo* = 'U', the array *ap* stores the packed upper triangular factor *U* of the factorization $A = PUDU^TP^T$.
If *uplo* = 'L', the array *ap* stores the packed lower triangular factor *L* of the factorization $A = PLDL^TP^T$.

n INTEGER. The order of matrix *A* ($n \geq 0$).

ap, work REAL for sspcon
DOUBLE PRECISION for dspcon
COMPLEX for cspcon
DOUBLE COMPLEX for zspcon.
Arrays: *ap*(*), *work*(*).

The array *ap* contains the packed factored matrix *A*, as returned by ?sptrf.
The dimension of *ap* must be at least max(1,*n*(*n*+1)/2).

The array *work* is a workspace for the routine.

The dimension of *work* must be at least max(1, 2*n).

ipiv INTEGER. Array, DIMENSION at least max(1,*n*).
The array *ipiv*, as returned by ?sptrf.

anorm REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
The norm of the *original* matrix *A* (see *Discussion*).

iwork — `INTEGER`.
Workspace array, `DIMENSION` at least max(1, *n*).

rwork — `REAL` for `cspcon`
`DOUBLE PRECISION` for `zspcon`
Workspace array, `DIMENSION` at least max(1, *n*).

Output Parameters

rcond — `REAL` for single precision flavors.
`DOUBLE PRECISION` for double precision flavors.
An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

The computed *rcond* is never less than ρ (the reciprocal of the true condition number) and in practice is nearly always less than 10ρ. A call to this routine involves solving a number of systems of linear equations $Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?hpcon

Estimates the reciprocal of the condition number of a packed Hermitian matrix.

Syntax

```
call chpcon ( uplo, n, ap, ipiv, anorm, rcond, work, rwork, info )
call zhpcon ( uplo, n, ap, ipiv, anorm, rcond, work, rwork, info )
```

Description

This routine estimates the reciprocal of the condition number of a Hermitian matrix A:

$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1$ (since A is Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \Sigma_i |a_{ij}|$ or $\|A\|_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?hptrf to compute the factorization of A.

Input Parameters

uplo — CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix A has been factored:

If *uplo* = 'U', the array *ap* stores the packed upper triangular factor U of the factorization $A = PUDU^TP^T$.

If *uplo* = 'L', the array *ap* stores the packed lower triangular factor L of the factorization $A = PLDL^TP^T$.

n — INTEGER. The order of matrix A ($n \geq 0$).

ap, work — COMPLEX for chpcon
DOUBLE COMPLEX for zhpcon.
Arrays: *ap*(*), *work*(*).

The array *ap* contains the packed factored matrix A, as returned by ?hptrf. The dimension of *ap* must be at least max(1,$n(n+1)/2$).

The array *work* is a workspace for the routine.
The dimension of *work* must be at least max(1, 2*n).

ipiv — INTEGER. Array, DIMENSION at least max(1,n).
The array *ipiv*, as returned by ?hptrf.

anorm — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
The norm of the *original* matrix A (see *Discussion*).

rwork — REAL for chpcon
DOUBLE PRECISION for zhpcon.
Workspace array, DIMENSION at least max(1, n).

Output Parameters

rcond — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

The computed *rcond* is never less than ρ (the reciprocal of the true condition number) and in practice is nearly always less than 10ρ. A call to this routine involves solving a number of systems of linear equations $Ax = b$; the number is usually 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

?trcon

Estimates the reciprocal of the condition number of a triangular matrix.

Syntax

```
call strcon (norm, uplo, diag, N, a, lda, rcond, work, iwork, info)
call dtrcon (norm, uplo, diag, N, a, lda, rcond, work, iwork, info)
call ctrcon (norm, uplo, diag, N, a, lda, rcond, work, rwork, info)
call ztrcon (norm, uplo, diag, N, a, lda, rcond, work, rwork, info)
```

Description

This routine estimates the reciprocal of the condition number of a triangular matrix A in either the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \, \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$\kappa_\infty(A) = ||A||_\infty \, ||A^{-1}||_\infty = \kappa_1(A^T) = \kappa_1(A^H)$.

Input Parameters

norm	CHARACTER*1. Must be '1' or 'O' or 'I'. If *norm* = '1' or 'O', then the routine estimates $\kappa_1(A)$. If *norm* = 'I', then the routine estimates $\kappa_\infty(A)$.
uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether *A* is upper or lower triangular: If *uplo* = 'U', the array *a* stores the upper triangle of *A*, other array elements are not referenced. If *uplo* = 'L', the array *a* stores the lower triangle of *A*, other array elements are not referenced.
diag	CHARACTER*1. Must be 'N' or 'U'. If *diag* = 'N', then *A* is not a unit triangular matrix. If *diag* = 'U', then *A* is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array *a*.
n	INTEGER. The order of the matrix *A* ($n \geq 0$).
a, work	REAL for strcon DOUBLE PRECISION for dtrcon COMPLEX for ctrcon DOUBLE COMPLEX for ztrcon. Arrays: *a*(*lda*,*), *work*(*). The array *a* contains the matrix *A*. The second dimension of *a* must be at least max(1,*n*). The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*).
rwork	REAL for ctrcon DOUBLE PRECISION for ztrcon. Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

rcond — `REAL` for single precision flavors.
`DOUBLE PRECISION` for double precision flavors.
An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

The computed *rcond* is never less than ρ (the reciprocal of the true condition number) and in practice is nearly always less than 10ρ. A call to this routine involves solving a number of systems of linear equations $Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors and $4n^2$ operations for complex flavors.

?tpcon

Estimates the reciprocal of the condition number of a packed triangular matrix.

Syntax

```
call stpcon (norm, uplo, diag, n, ap, rcond, work, iwork, info)
call dtpcon (norm, uplo, diag, n, ap, rcond, work, iwork, info)
call ctpcon (norm, uplo, diag, n, ap, rcond, work, rwork, info)
call ztpcon (norm, uplo, diag, n, ap, rcond, work, rwork, info)
```

Description

This routine estimates the reciprocal of the condition number of a packed triangular matrix *A* in either the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \, \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$
$$\kappa_\infty(A) = \|A\|_\infty \, \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H) .$$

Input Parameters

norm — CHARACTER*1. Must be '1' or 'O' or 'I'.
If *norm* = '1' or 'O', then the routine estimates $\kappa_1(A)$.
If *norm* = 'I', then the routine estimates $\kappa_\infty(A)$.

uplo — CHARACTER*1. Must be 'U' or 'L'.
Indicates whether A is upper or lower triangular:

If *uplo* = 'U', the array *ap* stores the upper triangle of A in packed form.

If *uplo* = 'L', the array *ap* stores the lower triangle of A in packed form.

diag — CHARACTER*1. Must be 'N' or 'U'.

If *diag* = 'N', then A is not a unit triangular matrix.

If *diag* = 'U', then A is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array *ap*.

n — INTEGER. The order of the matrix A ($n \ge 0$).

ap, work — REAL for stpcon
DOUBLE PRECISION for dtpcon
COMPLEX for ctpcon
DOUBLE COMPLEX for ztpcon.
Arrays: *ap*(*), *work*(*).

The array *ap* contains the packed matrix A.
The dimension of *ap* must be at least max(1,n(n+1)/2).
The array *work* is a workspace for the routine.

The dimension of *work* must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.

iwork — INTEGER.
Workspace array, DIMENSION at least max(1, n).

rwork — REAL for ctpcon
DOUBLE PRECISION for ztpcon
Workspace array, DIMENSION at least max(1, n).

Output Parameters

rcond — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working

precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The computed *rcond* is never less than ρ (the reciprocal of the true condition number) and in practice is nearly always less than 10ρ. A call to this routine involves solving a number of systems of linear equations $Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors and $4n^2$ operations for complex flavors.

?tbcon

Estimates the reciprocal of the condition number of a triangular band matrix.

Syntax

```
call stbcon (norm, uplo, diag, n, kd, ab, ldab, rcond, work, iwork, info)
call dtbcon (norm, uplo, diag, n, kd, ab, ldab, rcond, work, iwork, info)
call ctbcon (norm, uplo, diag, n, kd, ab, ldab, rcond, work, rwork, info)
call ztbcon (norm, uplo, diag, n, kd, ab, ldab, rcond, work, rwork, info)
```

Description

This routine estimates the reciprocal of the condition number of a triangular band matrix A in either the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \, \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$
$$\kappa_\infty(A) = \|A\|_\infty \, \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H) .$$

Input Parameters

norm CHARACTER*1. Must be '1' or 'O' or 'I'.
If *norm* = '1' or 'O', then the routine estimates $\kappa_1(A)$.
If *norm* = 'I', then the routine estimates $\kappa_\infty(A)$.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates whether A is upper or lower triangular:
If *uplo* = `'U'`, the array *ap* stores the upper triangle of A in packed form.
If *uplo* = `'L'`, the array *ap* stores the lower triangle of A in packed form.

diag — `CHARACTER*1`. Must be `'N'` or `'U'`.

If *diag* = `'N'`, then A is not a unit triangular matrix.

If *diag* = `'U'`, then A is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array *ab*.

n — `INTEGER`. The order of the matrix A ($n \geq 0$).

kd — `INTEGER`. The number of super-diagonals or sub-diagonals in the matrix A ($kd \geq 0$).

ab, work — `REAL` for `stbcon`
`DOUBLE PRECISION` for `dtbcon`
`COMPLEX` for `ctbcon`
`DOUBLE COMPLEX` for `ztbcon`.
Arrays: *ab*(*ldab*,*), *work*(*).

The array *ab* contains the band matrix A.
The second dimension of *ab* must be at least max(1,*n*)).
The array *work* is a workspace for the routine.
The dimension of *work* must be at least max(1, 3*n*) for real flavors and max(1, 2*n*) for complex flavors.

ldab — `INTEGER`. The first dimension of the array *ab*.
($ldab \geq kd + 1$).

iwork — `INTEGER`.
Workspace array, `DIMENSION` at least max(1, *n*).

rwork — `REAL` for `ctbcon`
`DOUBLE PRECISION` for `ztbcon`.
Workspace array, `DIMENSION` at least max(1, *n*).

Output Parameters

rcond — `REAL` for single precision flavors.
`DOUBLE PRECISION` for double precision flavors.
An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working

precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info — INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The computed *rcond* is never less than ρ (the reciprocal of the true condition number) and in practice is nearly always less than 10ρ. A call
to this routine involves solving a number of systems of linear equations
$Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n(kd + 1)$ floating-point operations for real flavors and $8n(kd + 1)$ operations for complex flavors.

Refining the Solution and Estimating Its Error

This section describes the LAPACK routines for refining the computed solution of a system of linear equations and estimating the solution error. You can call these routines after factorizing the matrix of the system of equations and computing the solution (see Routines for Matrix Factorization and Routines for Solving Systems of Linear Equations).

?gerfs

Refines the solution of a system of linear equations with a general matrix and estimates its error.

Syntax

```
call sgerfs (trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
    ferr, berr, work, iwork, info)
call dgerfs (trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
    ferr, berr, work, iwork, info)
call cgerfs (trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
    ferr, berr, work, rwork, info)
call zgerfs (trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
    ferr, berr, work, rwork, info)
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $AX = B$ or $A^TX = B$ or $A^HX = B$ with a general matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error* β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta\, |a_{ij}|,\ \ |\delta b_i|/|b_i| \le \beta\, |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine `?getrf`
- call the solver routine `?getrs`.

Input Parameters

trans — `CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`.
Indicates the form of the equations:
If *trans* = `'N'`, the system has the form $AX = B$.
If *trans* = `'T'`, the system has the form $A^TX = B$.
If *trans* = `'C'`, the system has the form $A^HX = B$.

n — `INTEGER`. The order of the matrix A ($n \ge 0$).

nrhs — `INTEGER`. The number of right-hand sides ($nrhs \ge 0$).

a,af,b,x,work — `REAL` for `sgerfs`
`DOUBLE PRECISION` for `dgerfs`
`COMPLEX` for `cgerfs`
`DOUBLE COMPLEX` for `zgerfs`.

Arrays:

a(*lda*,*) contains the original matrix A, as supplied to `?getrf`.

af(*ldaf*,*) contains the factored matrix A, as returned by `?getrf`.

b(*ldb*,*) contains the right-hand side matrix B.

x(*ldx*,*) contains the solution matrix X.

work (*) is a workspace array.

The second dimension of *a* and *af* must be at least max(1,*n*); the second dimension of *b* and *x* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.

lda — `INTEGER`. The first dimension of *a*; $lda \geq \max(1, n)$.

ldaf — `INTEGER`. The first dimension of *af*; $ldaf \geq \max(1, n)$.

ldb — `INTEGER`. The first dimension of *b*; $ldb \geq \max(1, n)$.

ldx — `INTEGER`. The first dimension of *x*; $ldx \geq \max(1, n)$.

ipiv — `INTEGER`.
Array, `DIMENSION` at least max(1,*n*).
The *ipiv* array, as returned by ?getrf.

iwork — `INTEGER`.
Workspace array, `DIMENSION` at least max(1, *n*).

rwork — `REAL` for `cgerfs`
`DOUBLE PRECISION` for `zgerfs`.
Workspace array, `DIMENSION` at least max(1, *n*).

Output Parameters

x — The refined solution matrix *X*.

ferr, berr — `REAL` for single precision flavors.
`DOUBLE PRECISION` for double precision flavors.
Arrays, `DIMENSION` at least max(1,*nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?gbrfs

Refines the solution of a system of linear equations with a general band matrix and estimates its error.

Syntax

```
call sgbrfs (trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv,  b, ldb,
    x, ldx, ferr, berr, work, iwork, info)
call dgbrfs (trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv,  b, ldb,
    x, ldx, ferr, berr, work, iwork, info)
call cgbrfs (trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv,  b, ldb,
    x, ldx, ferr, berr, work, rwork, info)
call zgbrfs (trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv,  b, ldb,
    x, ldx, ferr, berr, work, rwork, info)
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $AX = B$ or $A^T X = B$ or $A^H X = B$ with a band matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error* β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta\,|a_{ij}|$, $|\delta b_i|/|b_i| \le \beta\,|b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine `?gbtrf`
- call the solver routine `?gbtrs`.

Input Parameters

trans — CHARACTER*1. Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If *trans* = 'N', the system has the form $AX = B$.

If *trans* = 'T', the system has the form $A^T X = B$.

If *trans* = 'C', the system has the form $A^H X = B$.

n — INTEGER. The order of the matrix A ($n \geq 0$).

kl — INTEGER. The number of sub-diagonals within the band of A ($kl \geq 0$).

ku — INTEGER. The number of super-diagonals within the band of A ($ku \geq 0$).

nrhs — INTEGER. The number of right-hand sides ($nrhs \geq 0$).

ab, *afb*, *b*, *x*, *work* — REAL for sgbrfs
DOUBLE PRECISION for dgbrfs
COMPLEX for cgbrfs
DOUBLE COMPLEX for zgbrfs.

Arrays:

ab(*ldab*,*) contains the original band matrix A, as supplied to ?gbtrf, but stored in rows from 1 to $kl + ku + 1$.

afb(*ldafb*,*) contains the factored band matrix A, as returned by ?gbtrf.

b(*ldb*,*) contains the right-hand side matrix B.

x(*ldx*,*) contains the solution matrix X.

work (*) is a workspace array.

The second dimension of *ab* and *afb* must be at least max(1,*n*); the second dimension of *b* and *x* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.

ldab — INTEGER. The first dimension of *ab*.

ldafb — INTEGER. The first dimension of *afb* .

ldb — INTEGER. The first dimension of *b*; $ldb \geq \max(1, n)$.

ldx — INTEGER. The first dimension of *x*; $ldx \geq \max(1, n)$.

ipiv — INTEGER.
Array, DIMENSION at least max(1,*n*).
The *ipiv* array, as returned by ?gbtrf.

iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*).
rwork	REAL for cgbrfs DOUBLE PRECISION for zgbrfs Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

x	The refined solution matrix *X*.
ferr, berr	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1,*nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	INTEGER. If *info* =0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n(kl + ku)$ floating-point operations (for real flavors) or $16n(kl + ku)$ operations (for complex flavors). In addition, each step of iterative refinement involves $2n(4kl + 3ku)$ operations (for real flavors) or $8n(4kl + 3ku)$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?gtrfs

Refines the solution of a system of linear equations with a tridiagonal matrix and estimates its error.

Syntax

```
call sgtrfs (trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb,
    x, ldx, ferr, berr, work, iwork, info)
```

```
call dgtrfs (trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb,
    x, ldx, ferr, berr, work, iwork, info)
call cgtrfs (trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb,
    x, ldx, ferr, berr, work, rwork, info)
call zgtrfs (trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb,
    x, ldx, ferr, berr, work, rwork, info)
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $AX = B$ or $A^TX = B$ or $A^HX = B$ with a tridiagonal matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error* β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta\ |a_{ij}|,\ \ |\delta b_i|/|b_i| \le \beta\ |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine `?gttrf`
- call the solver routine `?gttrs`.

Input Parameters

trans — `CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`.
Indicates the form of the equations:
If *trans* = `'N'`, the system has the form $AX = B$.
If *trans* = `'T'`, the system has the form $A^TX = B$.
If *trans* = `'C'`, the system has the form $A^HX = B$.

n — `INTEGER`. The order of the matrix A ($n \ge 0$).

nrhs — `INTEGER`. The number of right-hand sides , i.e., the number of columns of the matrix B ($nrhs \ge 0$).

dl, d, du, dlf, df, duf, du2, b, x, work — `REAL` for `sgtrfs`
`DOUBLE PRECISION` for `dgtrfs`
`COMPLEX` for `cgtrfs`
`DOUBLE COMPLEX` for `zgtrfs`.

Arrays:

dl, dimension $(n - 1)$, contains the subdiagonal elements of A.

d, dimension (n), contains the diagonal elements of A.

du, dimension (n - 1), contains the superdiagonal elements of A.

dlf, dimension (n - 1), contains the (n - 1) multipliers that define the matrix L from the LU factorization of A as computed by ?gttrf.

df, dimension (n), contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A.

duf, dimension (n - 1), contains the (n - 1) elements of the first super-diagonal of U.

du2, dimension (n - 2), contains the (n - 2) elements of the second super-diagonal of U.

b(*ldb*,*nrhs*) contains the right-hand side matrix B.

x(*ldx*,*nrhs*) contains the solution matrix X, as computed by ?gttrs.

work (*) is a workspace array;
the dimension of *work* must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.

ldb — INTEGER. The first dimension of *b*; *ldb* ≥ max(1, n).

ldx — INTEGER. The first dimension of *x*; *ldx* ≥ max(1, n).

ipiv — INTEGER.
Array, DIMENSION at least max(1,n).
The *ipiv* array, as returned by ?gttrf.

iwork — INTEGER.
Workspace array, DIMENSION (n). Used for real flavors only.

rwork — REAL for cgtrfs
DOUBLE PRECISION for zgtrfs.
Workspace array, DIMENSION (n). Used for complex flavors only.

Output Parameters

x — The refined solution matrix X.

ferr, *berr* — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(1,*nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

?porfs

Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite matrix and estimates its error.

Syntax

```
call sporfs (uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr,
    work, iwork, info)
call dporfs (uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr,
    work, iwork, info)
call cporfs (uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr,
    work, rwork, info)
call zporfs (uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr,
    work, rwork, info)
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $AX = B$ with a symmetric (Hermitian) positive definite matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error* β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta\, |a_{ij}|,\ \ |\delta b_i|/|b_i| \le \beta\, |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine `?potrf`
- call the solver routine `?potrs`.

Input Parameters

uplo `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates how the input matrix A has been factored:

	If `uplo` = `'U'`, the array `af` stores the factor U of the Cholesky factorization $A = U^H U$. If `uplo` = `'L'`, the array `af` stores the factor L of the Cholesky factorization $A = LL^H$.
n	`INTEGER`. The order of the matrix A ($n \ge 0$).
nrhs	`INTEGER`. The number of right-hand sides ($nrhs \ge 0$).
a,af,b,x,work	`REAL` for `sporfs` `DOUBLE PRECISION` for `dporfs` `COMPLEX` for `cporfs` `DOUBLE COMPLEX` for `zporfs`. Arrays: `a(lda,*)` contains the original matrix A, as supplied to `?potrf`. `af(ldaf,*)` contains the factored matrix A, as returned by `?potrf`. `b(ldb,*)` contains the right-hand side matrix B. `x(ldx,*)` contains the solution matrix X. `work (*)` is a workspace array. The second dimension of `a` and `af` must be at least max(1,*n*); the second dimension of `b` and `x` must be at least max(1,*nrhs*); the dimension of `work` must be at least max(1, 3***n*) for real flavors and max(1, 2***n*) for complex flavors.
lda	`INTEGER`. The first dimension of `a`; $lda \ge$ max(1, *n*).
ldaf	`INTEGER`. The first dimension of `af`; $ldaf \ge$ max(1, *n*).
ldb	`INTEGER`. The first dimension of `b`; $ldb \ge$ max(1, *n*).
ldx	`INTEGER`. The first dimension of `x`; $ldx \ge$ max(1, *n*).
iwork	`INTEGER`. Workspace array, `DIMENSION` at least max(1, *n*).
rwork	`REAL` for `cporfs` `DOUBLE PRECISION` for `zporfs` Workspace array, `DIMENSION` at least max(1, *n*).

Output Parameters

x	The refined solution matrix X.

ferr, berr	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. Arrays, `DIMENSION` at least max(1,*nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?pprfs

Refines the solution of a system of linear equations with a packed symmetric (Hermitian) positive-definite matrix and estimates its error.

Syntax

```
call spprfs (uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work,
    iwork, info)
call dpprfs (uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work,
    iwork, info)
call cpprfs (uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work,
    rwork, info)
call zpprfs (uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work,
    rwork, info)
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $AX = B$ with a packed symmetric (Hermitian) positive definite matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error* β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta |a_{ij}|$, $|\delta b_i|/|b_i| \le \beta |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine `?pptrf`
- call the solver routine `?pptrs`.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates how the input matrix A has been factored:
If *uplo* = `'U'`, the array *afp* stores the packed factor U of the Cholesky factorization $A = U^H U$.
If *uplo* = `'L'`, the array *afp* stores the packed factor L of the Cholesky factorization $A = LL^H$.

n — `INTEGER`. The order of the matrix A ($n \ge 0$).

nrhs — `INTEGER`. The number of right-hand sides ($nrhs \ge 0$).

ap, afp, b, x, work — `REAL` for `spprfs`
`DOUBLE PRECISION` for `dpprfs`
`COMPLEX` for `cpprfs`
`DOUBLE COMPLEX` for `zpprfs`.

Arrays:

ap(*) contains the original packed matrix A, as supplied to `?pptrf`.

afp(*) contains the factored packed matrix A, as returned by `?pptrf`.

b(*ldb*, *) contains the right-hand side matrix B.

x(*ldx*, *) contains the solution matrix X.

work (*) is a workspace array.

	The dimension of arrays `ap` and `afp` must be at least max(1,n(n+1)/2); the second dimension of `b` and `x` must be at least max(1,`nrhs`); the dimension of `work` must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.
ldb	`INTEGER`. The first dimension of `b`; `ldb` ≥ max(1, n).
ldx	`INTEGER`. The first dimension of `x`; `ldx` ≥ max(1, n).
iwork	`INTEGER`. Workspace array, `DIMENSION` at least max(1, n).
rwork	`REAL` for `cpprfs` `DOUBLE PRECISION` for `zpprfs` Workspace array, `DIMENSION` at least max(1, n).

Output Parameters

x	The refined solution matrix X.
ferr, berr	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. Arrays, `DIMENSION` at least max(1,`nrhs`). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	`INTEGER`. If `info`=0, the execution is successful. If `info` = `-i`, the `i`th parameter had an illegal value.

Application Notes

The bounds returned in `ferr` are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $Ax = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?pbrfs

Refines the solution of a system of linear equations with a band symmetric (Hermitian) positive-definite matrix and estimates its error.

Syntax

```
call spbrfs (uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx,
    ferr, berr, work, iwork, info)
call dpbrfs (uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx,
    ferr, berr, work, iwork, info)
call cpbrfs (uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx,
    ferr, berr, work, rwork, info)
call zpbrfs (uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx,
    ferr, berr, work, rwork, info)
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $AX = B$ with a symmetric (Hermitian) positive definite band matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error* β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta\,|a_{ij}|,\ \ |\delta b_i|/|b_i| \le \beta\,|b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine `?pbtrf`
- call the solver routine `?pbtrs`.

Input Parameters

uplo `CHARACTER*1`. Must be `'U'` or `'L'`.

Indicates how the input matrix A has been factored:

If *uplo* = 'U', the array *afb* stores the factor U of the Cholesky factorization $A = U^H U$.
If *uplo* = 'L', the array *afb* stores the factor L of the Cholesky factorization $A = LL^H$.

n — INTEGER. The order of the matrix A ($n \geq 0$).

kd — INTEGER. The number of super-diagonals or sub-diagonals in the matrix A ($kd \geq 0$).

nrhs — INTEGER. The number of right-hand sides ($nrhs \geq 0$).

ab,afb,b,x,work — REAL for spbrfs
DOUBLE PRECISION for dpbrfs
COMPLEX for cpbrfs
DOUBLE COMPLEX for zpbrfs.

Arrays:

ab(*ldab*,*) contains the original band matrix A, as supplied to ?pbtrf.

afb(*ldafb*,*) contains the factored band matrix A, as returned by ?pbtrf.

b(*ldb*,*) contains the right-hand side matrix B.

x(*ldx*,*) contains the solution matrix X.

work (*) is a workspace array.

The second dimension of *ab* and *afb* must be at least max(1,*n*); the second dimension of *b* and *x* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.

ldab — INTEGER. The first dimension of *ab*; $ldab \geq kd + 1$.

ldafb — INTEGER. The first dimension of *afb*; $ldafb \geq kd + 1$.

ldb — INTEGER. The first dimension of *b*; $ldb \geq \max(1, n)$.

ldx — INTEGER. The first dimension of *x*; $ldx \geq \max(1, n)$.

iwork — INTEGER.
Workspace array, DIMENSION at least max(1, *n*).

rwork — REAL for cpbrfs
DOUBLE PRECISION for zpbrfs
Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

x — The refined solution matrix X.

ferr, berr	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. Arrays, `DIMENSION` at least max(1,*nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	`INTEGER`. If *info* =0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of 8*n*kd* floating-point operations (for real flavors) or 32*n*kd* operations (for complex flavors). In addition, each step of iterative refinement involves 12*n*kd* operations (for real flavors) or 48*n*kd* operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately 4*n*kd* floating-point operations for real flavors or 16*n*kd* for complex flavors.

?ptrfs

Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal matrix and estimates its error.

Syntax

```
call sptrfs (n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work,
    info)
call dptrfs (n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work,
    info)
call cptrfs (uplo, n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr,
    work, rwork, info)
call cptrfs (uplo, n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr,
    work, rwork, info)
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $AX = B$ with a symmetric (Hermitian) positive definite tridiagonal matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error* β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta\ |a_{ij}|$, $|\delta b_i|/|b_i| \le \beta\ |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?pttrf
- call the solver routine ?pttrs.

Input Parameters

uplo — CHARACTER*1. Used for complex flavors only.
Must be 'U' or 'L'.

Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix A is stored and how A is factored:
If *uplo* = 'U', the array *e* stores the superdiagonal of A, and A is factored as $U^H DU$;
If *uplo* = 'L', the array *e* stores the subdiagonal of A, and A is factored as LDL^H.

n — INTEGER. The order of the matrix A ($n \ge 0$).

nrhs — INTEGER. The number of right-hand sides ($nrhs \ge 0$).

d, df, rwork — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors
Arrays: *d*(*n*), *df*(*n*), *rwork*(*n*).
The array *d* contains the *n* diagonal elements of the tridiagonal matrix A.
The array *df* contains the *n* diagonal elements of the diagonal matrix D from the factorization of A as computed by ?pttrf.
The array *rwork* is a workspace array used for complex flavors only.

e, ef, b, x, work — REAL for sptrfs
DOUBLE PRECISION for dptrfs
COMPLEX for cptrfs
DOUBLE COMPLEX for zptrfs.
Arrays: *e*(*n* - 1), *ef*(*n* - 1), *b*(*ldb*,*nrhs*), *x*(*ldx*,*nrhs*), *work*(*).

The array *e* contains the $(n - 1)$ off-diagonal elements of the tridiagonal matrix A (see *uplo*).
The array *ef* contains the $(n - 1)$ off-diagonal elements of the unit bidiagonal factor U or L from the factorization computed by ?pttrf (see *uplo*).
The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations.
The array *x* contains the solution matrix X as computed by ?pttrs.
The array *work* is a workspace array. The dimension of *work* must be at least 2**n* for real flavors, and at least *n* for complex flavors.

ldb INTEGER. The leading dimension of *b*; *ldb* ≥ max(1, *n*).

ldx INTEGER. The leading dimension of *x*; *ldx* ≥ max(1, *n*).

Output Parameters

x The refined solution matrix X.

ferr, berr REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(1,*nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

?syrfs

Refines the solution of a system of linear equations with a symmetric matrix and estimates its error.

Syntax

```
call ssyrfs (uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
    berr, work, iwork, info)
call dsyrfs (uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
    berr, work, iwork, info)
call csyrfs (uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
    berr, work, rwork, info)
call zsyrfs (uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
    berr, work, rwork, info)
```

Discussion

This routine performs an iterative refinement of the solution to a system of linear equations $AX = B$ with a symmetric full-storage matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error* β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta\, |a_{ij}|$, $|\delta b_i|/|b_i| \le \beta\, |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine `?sytrf`
- call the solver routine `?sytrs`.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates how the input matrix A has been factored:
If *uplo* = `'U'`, the array *af* stores the Bunch-Kaufman factorization $A = PUDU^TP^T$.
If *uplo* = `'L'`, the array *af* stores the Bunch-Kaufman factorization $A = PLDL^TP^T$.

n — `INTEGER`. The order of the matrix A ($n \ge 0$).

nrhs — `INTEGER`. The number of right-hand sides (*nrhs* ≥ 0).

a,af,b,x,work — `REAL` for `ssyrfs`
`DOUBLE PRECISION` for `dsyrfs`
`COMPLEX` for `csyrfs`
`DOUBLE COMPLEX` for `zsyrfs`.

Arrays:

a(*lda*,*) contains the original matrix A, as supplied to `?sytrf`.

af(*ldaf*,*) contains the factored matrix A, as returned by `?sytrf`.

b(*ldb*,*) contains the right-hand side matrix B.

x(*ldx*,*) contains the solution matrix X.

work (*) is a workspace array.

The second dimension of *a* and *af* must be at least max(1,*n*); the second dimension of *b* and *x* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.

lda — INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).

ldaf — INTEGER. The first dimension of *af*; *ldaf* ≥ max(1, *n*).

ldb — INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

ldx — INTEGER. The first dimension of *x*; *ldx* ≥ max(1, *n*).

ipiv — INTEGER.
Array, DIMENSION at least max(1,*n*).
The *ipiv* array, as returned by ?sytrf.

iwork — INTEGER.
Workspace array, DIMENSION at least max(1, *n*).

rwork — REAL for csyrfs
DOUBLE PRECISION for zsyrfs.
Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

x — The refined solution matrix *X*.

ferr, berr — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(1,*nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?herfs

Refines the solution of a system of linear equations with a complex Hermitian matrix and estimates its error.

Syntax

```
call cherfs (uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
    berr, work, rwork, info)
call zherfs (uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
    berr, work, rwork, info)
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $AX = B$ with a complex Hermitian full-storage matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error* β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta\,|a_{ij}|,\ \ |\delta b_i|/|b_i| \le \beta\,|b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?hetrf
- call the solver routine ?hetrs.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If *uplo* = 'U', the array *af* stores the Bunch-Kaufman factorization $A = PUDU^HP^T$. If *uplo* = 'L', the array *af* stores the Bunch-Kaufman factorization $A = PLDL^HP^T$.
n	INTEGER. The order of the matrix A ($n \ge 0$).
nrhs	INTEGER. The number of right-hand sides ($nrhs \ge 0$).
a,af,b,x,work	COMPLEX for cherfs DOUBLE COMPLEX for zherfs.

Arrays:

a(lda,*) contains the original matrix *A*, as supplied to ?hetrf.

af(ldaf,*) contains the factored matrix *A*, as returned by ?hetrf.

b(ldb,*) contains the right-hand side matrix *B*.

x(ldx,*) contains the solution matrix *X*.

work (*) is a workspace array.

The second dimension of a and af must be at least max(1,*n*); the second dimension of b and x must be at least max(1,*nrhs*); the dimension of work must be at least max(1, 2*n).

lda INTEGER. The first dimension of a; lda ≥ max(1, *n*).

ldaf INTEGER. The first dimension of af; ldaf ≥ max(1, *n*).

ldb INTEGER. The first dimension of b; ldb ≥ max(1, *n*).

ldx INTEGER. The first dimension of x; ldx ≥ max(1, *n*).

ipiv INTEGER.
Array, DIMENSION at least max(1,*n*).
The ipiv array, as returned by ?hetrf.

rwork REAL for cherfs
DOUBLE PRECISION for zherfs.
Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

x The refined solution matrix *X*.

ferr, berr REAL for cherfs
DOUBLE PRECISION for zherfs.
Arrays, DIMENSION at least max(1,*nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.

info INTEGER.
If info = 0, the execution is successful.
If info = -i, the ith parameter had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $16n^2$ operations. In addition, each step of iterative refinement involves $24n^2$ operations; the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

The real counterpart of this routine is `ssyrfs` / `dsyrfs`.

?sprfs

Refines the solution of a system of linear equations with a packed symmetric matrix and estimates the solution error.

Syntax

```
call ssprfs (uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr,
    work, iwork, info)
call dsprfs (uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr,
    work, iwork, info)
call csprfs (uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr,
    work, rwork, info)
call zsprfs (uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr,
    work, rwork, info)
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $AX = B$ with a packed symmetric matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error* β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta\,|a_{ij}|$, $|\delta b_i|/|b_i| \le \beta\,|b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?sptrf
- call the solver routine ?sptrs.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix A has been factored:
If *uplo* = 'U', the array *afp* stores the packed Bunch-Kaufman factorization $A = PUDU^TP^T$.
If *uplo* = 'L', the array *afp* stores the packed Bunch-Kaufman factorization $A = PLDL^TP^T$.

n INTEGER. The order of the matrix A ($n \geq 0$).

nrhs INTEGER. The number of right-hand sides ($nrhs \geq 0$).

ap,afp,b,x,work REAL for ssprfs
DOUBLE PRECISION for dsprfs
COMPLEX for csprfs
DOUBLE COMPLEX for zsprfs.

Arrays:

ap(*) contains the original packed matrix A, as supplied to ?sptrf.

afp(*) contains the factored packed matrix A, as returned by ?sptrf.

b(*ldb*,*) contains the right-hand side matrix B.

x(*ldx*,*) contains the solution matrix X.

work (*) is a workspace array.

The dimension of arrays *ap* and *afp* must be at least max(1,*n*(*n*+1)/2); the second dimension of *b* and *x* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.

ldb INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

ldx INTEGER. The first dimension of *x*; *ldx* ≥ max(1, *n*).

ipiv INTEGER.
Array, DIMENSION at least max(1,*n*).
The *ipiv* array, as returned by ?sptrf.

iwork INTEGER.
Workspace array, DIMENSION at least max(1, *n*).

rwork REAL for csprfs
DOUBLE PRECISION for zsprfs
Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

x	The refined solution matrix *X*.
ferr, berr	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. Arrays, `DIMENSION` at least max(1,*nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $Ax = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?hprfs

Refines the solution of a system of linear equations with a packed complex Hermitian matrix and estimates the solution error.

Syntax

```
call chprfs (uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr,
    work, rwork, info)
call zhprfs (uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr,
    work, rwork, info)
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $AX = B$ with a packed complex Hermitian matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error* β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta\, |a_{ij}|,\ \ |\delta b_i|/|b_i| \le \beta\, |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?hptrf
- call the solver routine ?hptrs.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates how the input matrix A has been factored:
If *uplo* = `'U'`, the array *afp* stores the packed Bunch-Kaufman factorization $A = PUDU^HP^T$.
If *uplo* = `'L'`, the array *afp* stores the packed Bunch-Kaufman factorization $A = PLDL^HP^T$.

n — `INTEGER`. The order of the matrix A ($n \ge 0$).

nrhs — `INTEGER`. The number of right-hand sides (*nrhs* ≥ 0).

ap, afp, b, x, work — `COMPLEX` for `chprfs`
`DOUBLE COMPLEX` for `zhprfs`.

Arrays:

ap(*) contains the original packed matrix A, as supplied to ?hptrf.

afp(*) contains the factored packed matrix A, as returned by ?hptrf.

b(*ldb*, *) contains the right-hand side matrix B.

x(*ldx*, *) contains the solution matrix X.

work (*) is a workspace array.

The dimension of arrays *ap* and *afp* must be at least max(1,*n*(*n*+1)/2); the second dimension of *b* and *x* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1, 2***n*).

ldb — `INTEGER`. The first dimension of *b*; *ldb* ≥ max(1, *n*).

ldx INTEGER. The first dimension of *x*; *ldx* ≥ max(1, *n*).

ipiv INTEGER.
Array, DIMENSION at least max(1,*n*).
The *ipiv* array, as returned by ?hptrf.

rwork REAL for chprfs
DOUBLE PRECISION for zhprfs
Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

x The refined solution matrix *X*.

ferr, berr REAL for chprfs.
DOUBLE PRECISION for zhprfs.
Arrays, DIMENSION at least max(1,*nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $16n^2$ operations. In addition, each step of iterative refinement involves $24n^2$ operations; the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

The real counterpart of this routine is ssprfs / dsprfs.

?trrfs

Estimates the error in the solution of a system of linear equations with a triangular matrix.

Syntax

```
call strrfs (uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr,
    berr, work, iwork, info)
call dtrrfs (uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr,
    berr, work, iwork, info)
call ctrrfs (uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr,
    berr, work, rwork, info)
call ztrrfs (uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr,
    berr, work, rwork, info)
```

Description

This routine estimates the errors in the solution to a system of linear equations $AX = B$ or $A^TX = B$ or $A^HX = B$ with a triangular matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error* β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta\, |a_{ij}|$, $|\delta b_i|/|b_i| \le \beta\, |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

The routine also estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine `?trtrs`.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates whether A is upper or lower triangular:

If *uplo* = `'U'`, then A is upper triangular.
If *uplo* = `'L'`, then A is lower triangular.

trans — `CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`.
Indicates the form of the equations:
If *trans* = `'N'`, the system has the form $AX = B$.
If *trans* = `'T'`, the system has the form $A^TX = B$.

	If *trans* = 'C', the system has the form $A^H X = B$.
diag	CHARACTER*1. Must be 'N' or 'U'. If *diag* = 'N', then *A* is not a unit triangular matrix. If *diag* = 'U', then *A* is unit triangular: diagonal elements of *A* are assumed to be 1 and not referenced in the array *a*.
n	INTEGER. The order of the matrix *A* ($n \geq 0$).
nrhs	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
a, b, x, work	REAL for strrfs DOUBLE PRECISION for dtrrfs COMPLEX for ctrrfs DOUBLE COMPLEX for ztrrfs. Arrays: *a*(*lda*,*) contains the upper or lower triangular matrix *A*, as specified by *uplo*. *b*(*ldb*,*) contains the right-hand side matrix *B*. *x*(*ldx*,*) contains the solution matrix *X*. *work* (*) is a workspace array. The second dimension of *a* must be at least max(1,*n*); the second dimension of *b* and *x* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.
lda	INTEGER. The first dimension of *a*; $lda \geq \max(1, n)$.
ldb	INTEGER. The first dimension of *b*; $ldb \geq \max(1, n)$.
ldx	INTEGER. The first dimension of *x*; $ldx \geq \max(1, n)$.
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*).
rwork	REAL for ctrrfs DOUBLE PRECISION for ztrrfs Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

ferr, berr — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(1,*nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations $Ax = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors or $4n^2$ for complex flavors.

?tprfs

Estimates the error in the solution of a system of linear equations with a packed triangular matrix.

Syntax

```
call stprfs (uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr,
    work, iwork, info)
call dtprfs (uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr,
    work, iwork, info)
call ctprfs (uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr,
    work, rwork, info)
call ztprfs (uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr,
    work, rwork, info)
```

Description

This routine estimates the errors in the solution to a system of linear equations $AX = B$ or $A^T X = B$ or $A^H X = B$ with a packed triangular matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error* β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta\,|a_{ij}|,\ \ |\delta b_i|/|b_i| \le \beta\,|b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

The routine also estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine `?tptrs`.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Indicates whether A is upper or lower triangular: If *uplo* = `'U'`, then A is upper triangular. If *uplo* = `'L'`, then A is lower triangular.
trans	`CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`. Indicates the form of the equations: If *trans* = `'N'`, the system has the form $AX = B$. If *trans* = `'T'`, the system has the form $A^T X = B$. If *trans* = `'C'`, the system has the form $A^H X = B$.
diag	`CHARACTER*1`. Must be `'N'` or `'U'`. If *diag* = `'N'`, A is not a unit triangular matrix. If *diag* = `'U'`, A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array *ap*.
n	`INTEGER`. The order of the matrix A ($n \ge 0$).
nrhs	`INTEGER`. The number of right-hand sides ($nrhs \ge 0$).
ap, b, x, work	`REAL` for `strrfs` `DOUBLE PRECISION` for `dtrrfs` `COMPLEX` for `ctrrfs` `DOUBLE COMPLEX` for `ztrrfs`. Arrays: *ap*(*) contains the upper or lower triangular matrix A, as specified by *uplo*. *b*(*ldb*,*) contains the right-hand side matrix B. *x*(*ldx*,*) contains the solution matrix X.

work (*) is a workspace array.

The dimension of *ap* must be at least max(1,*n*(*n*+1)/2); the second dimension of *b* and *x* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.

ldb — INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

ldx — INTEGER. The first dimension of *x*; *ldx* ≥ max(1, *n*).

iwork — INTEGER.
Workspace array, DIMENSION at least max(1, *n*).

rwork — REAL for ctrrfs
DOUBLE PRECISION for ztrrfs
Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

ferr, berr — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(1,*nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations $Ax = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors or $4n^2$ for complex flavors.

?tbrfs

Estimates the error in the solution of a system of linear equations with a triangular band matrix.

Syntax

```
call stbrfs (uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx,
    ferr, berr, work, iwork, info)
call dtbrfs (uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx,
    ferr, berr, work, iwork, info)
call ctbrfs (uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx,
    ferr, berr, work, rwork, info)
call ztbrfs (uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx,
    ferr, berr, work, rwork, info)
```

Description

This routine estimates the errors in the solution to a system of linear equations $AX = B$ or $A^TX = B$ or $A^HX = B$ with a triangular band matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the *component-wise backward error* β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta\ |a_{ij}|,\ \ |\delta b_i|/|b_i| \le \beta\ |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

The routine also estimates the *component-wise forward error* in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine `?tbtrs`.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates whether A is upper or lower triangular:

If *uplo* = 'U', then A is upper triangular.
If *uplo* = 'L', then A is lower triangular.

trans CHARACTER*1. Must be 'N' or 'T' or 'C'.
Indicates the form of the equations:
If *trans* = 'N', the system has the form $AX = B$.
If *trans* = 'T', the system has the form $A^TX = B$.
If *trans* = 'C', the system has the form $A^HX = B$.

diag CHARACTER*1. Must be 'N' or 'U'.
If *diag* = 'N', A is not a unit triangular matrix.

If *diag* = 'U', A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array *ab*.

n INTEGER. The order of the matrix A ($n \geq 0$).

kd INTEGER. The number of super-diagonals or sub-diagonals in the matrix A ($kd \geq 0$).

nrhs INTEGER. The number of right-hand sides ($nrhs \geq 0$).

ab, b, x, work REAL for stbrfs
DOUBLE PRECISION for dtbrfs
COMPLEX for ctbrfs
DOUBLE COMPLEX for ztbrfs.

Arrays:

ab(*ldab*,*) contains the upper or lower triangular matrix A, as specified by *uplo*, in band storage format.

b(*ldb*,*) contains the right-hand side matrix B.

x(*ldx*,*) contains the solution matrix X.

work (*) is a workspace array.

The second dimension of *a* must be at least max(1,*n*);
the second dimension of *b* and *x* must be at least max(1,*nrhs*).
The dimension of *work* must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.

ldab INTEGER. The first dimension of the array *ab*.
($ldab \geq kd + 1$).

ldb INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

ldx INTEGER. The first dimension of *x*; *ldx* ≥ max(1, *n*).

iwork INTEGER.
Workspace array, DIMENSION at least max(1, *n*).

rwork REAL for ctbrfs
DOUBLE PRECISION for ztbrfs
Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

ferr, berr REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(1,*nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations $Ax = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately 2*n***kd* floating-point operations for real flavors or 8*n***kd* operations for complex flavors.

Routines for Matrix Inversion

It is seldom necessary to compute an explicit inverse of a matrix.
In particular, do not attempt to solve a system of equations $Ax = b$ by first computing A^{-1} and then forming the matrix-vector product $x = A^{-1}b$.
Call a solver routine instead (see Routines for Solving Systems of Linear Equations); this is more efficient and more accurate.

However, matrix inversion routines are provided for the rare occasions when an explicit inverse matrix is needed.

?getri

Computes the inverse of an LU-factored general matrix.

Syntax

```
call sgetri (n, a, lda, ipiv, work, lwork, info)
call dgetri (n, a, lda, ipiv, work, lwork, info)
call cgetri (n, a, lda, ipiv, work, lwork, info)
call zgetri (n, a, lda, ipiv, work, lwork, info)
```

Description

This routine computes the inverse (A^{-1}) of a general matrix A.
Before calling this routine, call ?getrf to factorize A.

Input Parameters

n — INTEGER. The order of the matrix A ($n \geq 0$).

a, work — REAL for sgetri
DOUBLE PRECISION for dgetri
COMPLEX for cgetri
DOUBLE COMPLEX for zgetri.
Arrays: a(*lda*,*), work(*lwork*).
a(*lda*,*) contains the factorization of the matrix A, as returned by ?getrf: $A = PLU$.
The second dimension of *a* must be at least max(1,*n*).

work(*lwork*) is a workspace array.

lda	`INTEGER`. The first dimension of *a*; $lda \geq \max(1, n)$.
ipiv	`INTEGER`. Array, `DIMENSION` at least max(1,*n*). The *ipiv* array, as returned by `?getrf`.
lwork	`INTEGER`. The size of the *work* array ($lwork \geq n$) See *Application notes* for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the *n* by *n* matrix A^{-1}.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i*, the *i*th diagonal element of the factor *U* is zero, *U* is singular, and the inversion could not be completed.

Application Notes

For better performance, try using *lwork* = *n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The computed inverse *X* satisfies the following error bound:

$$|XA - I| \leq c(n)\varepsilon |X|P|L||U|$$

where $c(n)$ is a modest linear function of n; ε is the machine precision;
I denotes the identity matrix; *P*, *L*, and *U* are the factors of the matrix factorization *A* = *PLU*.

The total number of floating-point operations is approximately $(4/3)n^3$ for real flavors and $(16/3)n^3$ for complex flavors.

?potri

Computes the inverse of a symmetric (Hermitian) positive-definite matrix.

Syntax

```
call spotri (uplo, n, a, lda, info)
call dpotri (uplo, n, a, lda, info)
call cpotri (uplo, n, a, lda, info)
call zpotri (uplo, n, a, lda, info)
```

Discussion

This routine computes the inverse (A^{-1}) of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix A.
Before calling this routine, call ?potrf to factorize A.

Input Parameters

uplo — CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix A has been factored:
If *uplo* = 'U', the array *a* stores the factor U of the Cholesky factorization $A = U^H U$.
If *uplo* = 'L', the array *a* stores the factor L of the Cholesky factorization $A = LL^H$.

n — INTEGER. The order of the matrix A ($n \ge 0$).

a — REAL for spotri
DOUBLE PRECISION for dpotri
COMPLEX for cpotri
DOUBLE COMPLEX for zpotri.
Array: *a*(*lda*,*).

Contains the factorization of the matrix A, as returned by ?potrf.

The second dimension of *a* must be at least max(1,*n*).

lda — INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).

Output Parameters

a Overwritten by the *n* by *n* matrix A^{-1}.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, the *i*th diagonal element of the Cholesky factor (and hence the factor itself) is zero, and the inversion could not be completed.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$\|XA - I\|_2 \le c(n)\varepsilon\kappa_2(A), \quad \|AX - I\|_2 \le c(n)\varepsilon\kappa_2(A)$$

where $c(n)$ is a modest linear function of *n*, and ε is the machine precision; *I* denotes the identity matrix.

The 2-norm $\|A\|_2$ of a matrix *A* is defined by $\|A\|_2 = \max_{x \cdot x = 1}(Ax \cdot Ax)^{1/2}$, and the condition number $\kappa_2(A)$ is defined by $\kappa_2(A) = \|A\|_2 \, \|A^{-1}\|_2$.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

?pptri

Computes the inverse of a packed symmetric (Hermitian) positive-definite matrix

Syntax

```
call spptri (uplo, n, ap, info)
call dpptri (uplo, n, ap, info)
call cpptri (uplo, n, ap, info)
call zpptri (uplo, n, ap, info)
```

Description

This routine computes the inverse (A^{-1}) of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix A in *packed* form. Before calling this routine, call ?pptrf to factorize A.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix A has been factored:
If *uplo* = 'U', the array *ap* stores the packed factor U of the Cholesky factorization $A = U^H U$.
If *uplo* = 'L', the array *ap* stores the packed factor L of the Cholesky factorization $A = LL^H$.

n INTEGER. The order of the matrix A ($n \geq 0$).

ap REAL for spptri
DOUBLE PRECISION for dpptri
COMPLEX for cpptri
DOUBLE COMPLEX for zpptri.
Array, DIMENSION at least max(1,n(n+1)/2).

Contains the factorization of the packed matrix A, as returned by ?pptrf.

The dimension *ap* must be at least max(1,n(n+1)/2).

Output Parameters

ap Overwritten by the packed n by n matrix A^{-1}.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, the *i*th diagonal element of the Cholesky factor (and hence the factor itself) is zero, and the inversion could not be completed.

Application Notes

The computed inverse X satisfies the following error bounds:

$$\|XA - I\|_2 \leq c(n)\varepsilon\kappa_2(A), \quad \|AX - I\|_2 \leq c(n)\varepsilon\kappa_2(A)$$

where $c(n)$ is a modest linear function of n, and ε is the machine precision; I denotes the identity matrix.

The 2-norm $||A||_2$ of a matrix A is defined by $||A||_2 = \max_{x \cdot x=1}(Ax \cdot Ax)^{1/2}$, and the condition number $\kappa_2(A)$ is defined by $\kappa_2(A) = ||A||_2 \, ||A^{-1}||_2$.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

?sytri

Computes the inverse of a symmetric matrix.

Syntax

```
call ssytri (uplo, n, a, lda, ipiv, work, info)
call dsytri (uplo, n, a, lda, ipiv, work, info)
call csytri (uplo, n, a, lda, ipiv, work, info)
call zsytri (uplo, n, a, lda, ipiv, work, info)
```

Description

This routine computes the inverse (A^{-1}) of a symmetric matrix A.
Before calling this routine, call ?sytrf to factorize A.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates how the input matrix A has been factored:
If *uplo* = `'U'`, the array *a* stores the Bunch-Kaufman factorization $A = PUDU^TP^T$.
If *uplo* = `'L'`, the array *a* stores the Bunch-Kaufman factorization $A = PLDL^TP^T$.

n — `INTEGER`. The order of the matrix A ($n \geq 0$).

a, work — `REAL` for `ssytri`
`DOUBLE PRECISION` for `dsytri`
`COMPLEX` for `csytri`
`DOUBLE COMPLEX` for `zsytri`.
Arrays:

	`a(lda,*)` contains the factorization of the matrix A, as returned by `?sytrf`. The second dimension of `a` must be at least max(1,n).
	`work(*)` is a workspace array. The dimension of `work` must be at least max(1,2*n).
lda	`INTEGER`. The first dimension of `a`; `lda` ≥ max(1, n).
ipiv	`INTEGER`. Array, `DIMENSION` at least max(1,n). The `ipiv` array, as returned by `?sytrf`.

Output Parameters

a	Overwritten by the n by n matrix A^{-1}.
info	`INTEGER`. If `info` = 0, the execution is successful. If `info` =-`i`, the `i`th parameter had an illegal value. If `info` = `i`, the `i`th diagonal element of D is zero, D is singular, and the inversion could not be completed.

Application Notes

The computed inverse X satisfies the following error bounds:

$$|DU^TP^TXPU - I| \le c(n)\varepsilon\,(|D||U^T|P^T|X|P|U| + |D||D^{-1}|)$$

for *uplo* = `'U'`, and

$$|DL^TP^TXPL - I| \le c(n)\varepsilon\,(|D||L^T|P^T|X|P|L| + |D||D^{-1}|)$$

for *uplo* = `'L'`. Here $c(n)$ is a modest linear function of n, and ε is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

?hetri

Computes the inverse of a complex Hermitian matrix.

Syntax

```
call chetri (uplo, n, a, lda, ipiv, work, info)
call zhetri (uplo, n, a, lda, ipiv, work, info)
```

Description

This routine computes the inverse (A^{-1}) of a complex Hermitian matrix A. Before calling this routine, call ?hetrf to factorize A.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If *uplo* = 'U', the array *a* stores the Bunch-Kaufman factorization $A = PUDU^HP^T$. If *uplo* = 'L', the array *a* stores the Bunch-Kaufman factorization $A = PLDL^HP^T$.
n	INTEGER. The order of the matrix A ($n \geq 0$).
a, work	COMPLEX for chetri DOUBLE COMPLEX for zhetri. Arrays: *a*(*lda*,*) contains the factorization of the matrix A, as returned by ?hetrf. The second dimension of *a* must be at least max(1,*n*). *work*(*) is a workspace array. The dimension of *work* must be at least max(1,*n*).
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).
ipiv	INTEGER. Array, DIMENSION at least max(1,*n*). The *ipiv* array, as returned by ?hetrf.

Output Parameters

a Overwritten by the *n* by *n* matrix A^{-1}.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, the *i*th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|DU^H P^T XPU - I| \le c(n)\varepsilon (|D||U^H|P^T|X|P|U| + |D||D^{-1}|)$$

for *uplo* = 'U', and

$$|DL^H P^T XPL - I| \le c(n)\varepsilon (|D||L^H|P^T|X|P|L| + |D||D^{-1}|)$$

for *uplo* = 'L'. Here $c(n)$ is a modest linear function of *n*, and ε is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

The real counterpart of this routine is ?sytri.

?sptri

Computes the inverse of a symmetric matrix using packed storage.

Syntax

```
call ssptri (uplo, n, ap, ipiv, work, info)
call dsptri (uplo, n, ap, ipiv, work, info)
call csptri (uplo, n, ap, ipiv, work, info)
call zsptri (uplo, n, ap, ipiv, work, info)
```

Description

This routine computes the inverse (A^{-1}) of a packed symmetric matrix A.
Before calling this routine, call ?sptrf to factorize A.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If *uplo* = 'U', the array *ap* stores the Bunch-Kaufman factorization $A = PUDU^TP^T$. If *uplo* = 'L', the array *ap* stores the Bunch-Kaufman factorization $A = PLDL^TP^T$.
n	INTEGER. The order of the matrix A ($n \geq 0$).
ap, work	REAL for ssptri DOUBLE PRECISION for dsptri COMPLEX for csptri DOUBLE COMPLEX for zsptri. Arrays: *ap*(*) contains the factorization of the matrix A, as returned by ?sptrf. The dimension of *ap* must be at least max(1,*n*(*n*+1)/2). *work*(*) is a workspace array. The dimension of *work* must be at least max(1,*n*).
ipiv	INTEGER. Array, DIMENSION at least max(1,*n*). The *ipiv* array, as returned by ?sptrf.

Output Parameters

ap	Overwritten by the *n* by *n* matrix A^{-1} in packed form.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i*, the *i*th diagonal element of D is zero, D is singular, and the inversion could not be completed.

Application Notes

The computed inverse X satisfies the following error bounds:

$$|DU^TP^TXPU - I| \le c(n)\varepsilon(|D||U^T|P^T|X|P|U| + |D||D^{-1}|)$$

for *uplo* = `'U'`, and

$$|DL^TP^TXPL - I| \le c(n)\varepsilon(|D||L^T|P^T|X|P|L| + |D||D^{-1}|)$$

for *uplo* = `'L'`. Here $c(n)$ is a modest linear function of n, and ε is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

?hptri

Computes the inverse of a complex Hermitian matrix using packed storage.

Syntax

```
call chptri (uplo, n, ap, ipiv, work, info)
call zhptri (uplo, n, ap, ipiv, work, info)
```

Description

This routine computes the inverse (A^{-1}) of a complex Hermitian matrix A using packed storage. Before calling this routine, call `?hptrf` to factorize A.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates how the input matrix A has been factored:
If *uplo* = `'U'`, the array *ap* stores the packed Bunch-Kaufman factorization $A = PUDU^HP^T$.
If *uplo* = `'L'`, the array *ap* stores the packed Bunch-Kaufman factorization $A = PLDL^HP^T$.

n — `INTEGER`. The order of the matrix A ($n \ge 0$).

ap — `COMPLEX` for `chptri`
`DOUBLE COMPLEX` for `zhptri`.
Arrays:

ap(*) contains the factorization of the matrix *A*, as returned by `?hptrf`.
The dimension of *ap* must be at least max(1,*n*(*n*+1)/2).

work(*) is a workspace array.
The dimension of *work* must be at least max(1,*n*).

ipiv — `INTEGER`.
Array, `DIMENSION` at least max(1,*n*).
The *ipiv* array, as returned by `?hptrf`.

Output Parameters

ap — Overwritten by the *n* by *n* matrix A^{-1}.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, the *i*th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|DU^H P^T XPU - I| \le c(n)\varepsilon\,(|D||U^H|P^T|X|P|U| + |D||D^{-1}|)$$

for *uplo* = 'U', and

$$|DL^H P^T XPL - I| \le c(n)\varepsilon\,(|D||L^H|P^T|X|P|L| + |D||D^{-1}|)$$

for *uplo* = 'L'. Here *c*(*n*) is a modest linear function of *n*, and ε is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

The real counterpart of this routine is `?sptri`.

?trtri

Computes the inverse of a triangular matrix.

Syntax

```
call strtri (uplo, diag, n, a, lda, info)
call dtrtri (uplo, diag, n, a, lda, info)
call ctrtri (uplo, diag, n, a, lda, info)
call ztrtri (uplo, diag, n, a, lda, info)
```

Description

This routine computes the inverse (A^{-1}) of a triangular matrix A.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates whether A is upper or lower triangular:

If *uplo* = `'U'`, then A is upper triangular.
If *uplo* = `'L'`, then A is lower triangular.

diag — `CHARACTER*1`. Must be `'N'` or `'U'`.
If *diag* = `'N'`, then A is not a unit triangular matrix.

If *diag* = `'U'`, A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array *a*.

n — `INTEGER`. The order of the matrix A ($n \geq 0$).

a — `REAL` for `strtri`
`DOUBLE PRECISION` for `dtrtri`
`COMPLEX` for `ctrtri`
`DOUBLE COMPLEX` for `ztrtri`.

Array: `DIMENSION (lda,*)`.
Contains the matrix A.
The second dimension of *a* must be at least max(1,*n*).

lda — `INTEGER`. The first dimension of *a*; *lda* ≥ max(1, *n*).

Output Parameters

a Overwritten by the *n* by *n* matrix A^{-1}.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, the *i*th diagonal element of *A* is zero, *A* is singular, and the inversion could not be completed.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|XA - I| \leq c(n)\varepsilon |X||A|$$

$$|X - A^{-1}| \leq c(n)\varepsilon |A^{-1}||A||X|$$

where $c(n)$ is a modest linear function of n; ε is the machine precision;
I denotes the identity matrix.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

?tptri

Computes the inverse of a triangular matrix using packed storage.

Syntax

```
call stptri (uplo, diag, n, ap, info)
call dtptri (uplo, diag, n, ap, info)
call ctptri (uplo, diag, n, ap, info)
call ztptri (uplo, diag, n, ap, info)
```

Description

This routine computes the inverse (A^{-1}) of a packed triangular matrix *A*.

Input Parameters

uplo — CHARACTER*1. Must be 'U' or 'L'.
Indicates whether A is upper or lower triangular:

If *uplo* = 'U', then A is upper triangular.
If *uplo* = 'L', then A is lower triangular.

diag — CHARACTER*1. Must be 'N' or 'U'.

If *diag* = 'N', then A is not a unit triangular matrix.

If *diag* = 'U', A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array *ap*.

n — INTEGER. The order of the matrix A ($n \geq 0$).

ap — REAL for stptri
DOUBLE PRECISION for dtptri
COMPLEX for ctptri
DOUBLE COMPLEX for ztptri.

Array: DIMENSION at least max(1,$n(n+1)/2$).
Contains the packed triangular matrix A.

Output Parameters

ap — Overwritten by the packed n by n matrix A^{-1}.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, the *i*th diagonal element of A is zero, A is singular, and the inversion could not be completed.

Application Notes

The computed inverse X satisfies the following error bounds:

$$|XA - I| \leq c(n)\varepsilon |X||A|$$

$$|X - A^{-1}| \leq c(n)\varepsilon |A^{-1}||A||X|$$

where $c(n)$ is a modest linear function of n; ε is the machine precision;
I denotes the identity matrix.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

Routines for Matrix Equilibration

Routines described in this section are used to compute scaling factors needed to equilibrate a matrix. Note that these routines do not actually scale the matrices.

?geequ

Computes row and column scaling factors intended to equilibrate a matrix and reduce its condition number.

Syntax

```
call sgeequ (m, n, a, lda, r, c, rowcnd, colcnd, amax, info)
call dgeequ (m, n, a, lda, r, c, rowcnd, colcnd, amax, info)
call cgeequ (m, n, a, lda, r, c, rowcnd, colcnd, amax, info)
call zgeequ (m, n, a, lda, r, c, rowcnd, colcnd, amax, info)
```

Description

This routine computes row and column scalings intended to equilibrate an *m*-by-*n* matrix *A* and reduce its condition number. The output array *r* returns the row scale factors and the array *c* the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements $b_{ij}=r(i)*a_{ij}*c(j)$ have absolute value 1.

Input Parameters

m	INTEGER. The number of rows of the matrix A, $m \geq 0$.
n	INTEGER. The number of columns of the matrix A, $n \geq 0$.
a	REAL for sgeequ DOUBLE PRECISION for dgeequ COMPLEX for cgeequ DOUBLE COMPLEX for zgeequ. Array: DIMENSION (*lda*,*). Contains the *m*-by-*n* matrix *A* whose equilibration factors are to be computed. The second dimension of *a* must be at least max(1,*n*).
lda	INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *m*).

Output Parameters

r, c — REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
Arrays: *r*(*m*), *c*(*n*).
If *info* = 0, or *info* > *m*, the array *r* contains the row scale factors of the matrix *A*.
If *info* = 0 , the array *c* contains the column scale factors of the matrix *A*.

rowcnd — REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
If *info* = 0 or *info* > *m*, *rowcnd* contains the ratio of the smallest *r*(i) to the largest *r*(i).

colcnd — REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
If *info* = 0, *colcnd* contains the ratio of the smallest *c*(i) to the largest *c*(i).

amax — REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
Absolute value of the largest element of the matrix *A*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.
If *info* = *i* and
$i \le m$, the *i*th row of *A* is exactly zero;
$i > m$, the (i-m)th column of *A* is exactly zero.

Application Notes

All the components of *r* and *c* are restricted to be between SMLNUM = smallest safe number and BIGNUM = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of *A* but works well in practice.

If *rowcnd* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *r*. If *colcnd* ≥ 0.1 , it is not worth scaling by *c*.

If *amax* is very close to overflow or very close to underflow, the matrix *A* should be scaled.

?gbequ

Computes row and column scaling factors intended to equilibrate a band matrix and reduce its condition number.

Syntax

```
call sgbequ (m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info)
call dgbequ (m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info)
call cgbequ (m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info)
call zgbequ (m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info)
```

Description

This routine computes row and column scalings intended to equilibrate an *m*-by-*n* band matrix *A* and reduce its condition number. The output array *r* returns the row scale factors and the array *c* the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements $b_{ij}=r(i)*a_{ij}*c(j)$ have absolute value 1.

Input Parameters

m — `INTEGER`. The number of rows of the matrix A, $m \geq 0$.

n — `INTEGER`. The number of columns of the matrix A, $n \geq 0$.

kl — `INTEGER`. The number of sub-diagonals within the band of *A* ($kl \geq 0$).

ku — `INTEGER`. The number of super-diagonals within the band of *A* ($ku \geq 0$).

ab — `REAL` for `sgbequ`
`DOUBLE PRECISION` for `dgbequ`
`COMPLEX` for `cgbequ`
`DOUBLE COMPLEX` for `zgbequ`.

Array, `DIMENSION (`*ldab*`,*)`.
Contains the original band matrix *A* stored in rows from 1 to $kl + ku + 1$.

The second dimension of *ab* must be at least max(1,*n*);

ldab — `INTEGER`. The leading dimension of *ab*, $ldab \ge kl+ku+1$.

Output Parameters

r, c — `REAL` for single precision flavors;
`DOUBLE PRECISION` for double precision flavors.
Arrays: *r*(*m*), *c*(*n*).
If *info* = 0, or *info* > *m*, the array *r* contains the row scale factors of the matrix *A*.
If *info* = 0 , the array *c* contains the column scale factors of the matrix *A*.

rowcnd — `REAL` for single precision flavors;
`DOUBLE PRECISION` for double precision flavors.
If *info* = 0 or *info* > *m*, *rowcnd* contains the ratio of the smallest *r*(i) to the largest *r*(i).

colcnd — `REAL` for single precision flavors;
`DOUBLE PRECISION` for double precision flavors.
If *info* = 0, *colcnd* contains the ratio of the smallest *c*(i) to the largest *c*(i).

amax — `REAL` for single precision flavors;
`DOUBLE PRECISION` for double precision flavors.
Absolute value of the largest element of the matrix *A*.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i* and
$i \le m$, the *i*th row of *A* is exactly zero;
$i > m$, the (*i*-*m*)th column of *A* is exactly zero.

Application Notes

All the components of *r* and *c* are restricted to be between SMLNUM = smallest safe number and BIGNUM = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of *A* but works well in practice.

If $rowcnd \ge 0.1$ and *amax* is neither too large nor too small, it is not worth scaling by *r*. If $colcnd \ge 0.1$, it is not worth scaling by *c*.

If *amax* is very close to overflow or very close to underflow, the matrix *A* should be scaled.

?poequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix and reduce its condition number.

Syntax

```
call spoequ (n, a, lda, s, scond, amax, info)
call dpoequ (n, a, lda, s, scond, amax, info)
call cpoequ (n, a, lda, s, scond, amax, info)
call zpoequ (n, a, lda, s, scond, amax, info)
```

Description

This routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite matrix *A* and reduce its condition number (with respect to the two-norm). The output array `s` returns scale factors computed as

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled matrix B with elements b_{ij}=s(i)*a_{ij}*s(j) has diagonal elements equal to 1.

This choice of `s` puts the condition number of *B* within a factor *n* of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

n	`INTEGER`. The order of the matrix A, *n* ≥0.
a	`REAL` for `spoequ` `DOUBLE PRECISION` for `dpoequ` `COMPLEX` for `cpoequ` `DOUBLE COMPLEX` for `zpoequ`.

Array: `DIMENSION (lda,*)`.
Contains the *n*-by-*n* symmetric or Hermitian positive definite matrix *A* whose scaling factors are to be computed. Only diagonal elements of *A* are referenced.
The second dimension of *a* must be at least max(1,*n*).

lda — `INTEGER`. The leading dimension of *a*; *lda* ≥ max(1, *m*).

Output Parameters

s — `REAL` for single precision flavors;
`DOUBLE PRECISION` for double precision flavors.
Array, `DIMENSION (n)`.
If *info* = 0, the array *s* contains the scale factors for *A*.

scond — `REAL` for single precision flavors;
`DOUBLE PRECISION` for double precision flavors.
If *info* = 0, *scond* contains the ratio of the smallest *s*(i) to the largest *s*(i).

amax — `REAL` for single precision flavors;
`DOUBLE PRECISION` for double precision flavors.
Absolute value of the largest element of the matrix *A*.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.
If *info* = *i*, the *i*th diagonal element of *A* is nonpositive.

Application Notes

If *scond* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *s*.

If *amax* is very close to overflow or very close to underflow, the matrix *A* should be scaled.

?ppequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix in packed storage and reduce its condition number.

Syntax

```
call sppequ (uplo, n, ap, s, scond, amax, info)
call dppequ (uplo, n, ap, s, scond, amax, info)
call cppequ (uplo, n, ap, s, scond, amax, info)
call zppequ (uplo, n, ap, s, scond, amax, info)
```

Description

This routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite matrix *A* in packed storage and reduce its condition number (with respect to the two-norm). The output array *s* returns scale factors computed as

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled matrix B with elements $b_{ij}=s(i)*a_{ij}*s(j)$ has diagonal elements equal to 1.

This choice of *s* puts the condition number of *B* within a factor *n* of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

uplo — CHARACTER*1. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of *A* is packed in the array *ap*:
If *uplo* = 'U', the array *ap* stores the upper triangular part of the matrix *A*.
If *uplo* = 'L', the array *ap* stores the lower triangular part of the matrix *A*.

n — INTEGER. The order of matrix *A* ($n \ge 0$).

ap — REAL for sppequ
DOUBLE PRECISION for dppequ
COMPLEX for cppequ

DOUBLE COMPLEX for zppequ.
Array, DIMENSION at least max(1,n(n+1)/2).
The array *ap* contains either the upper or the lower triangular part of the matrix *A* (as specified by *uplo*) in *packed storage* (see Matrix Storage Schemes).

Output Parameters

s — REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
Array, DIMENSION (*n*).
If *info* = 0, the array *s* contains the scale factors for *A*.

scond — REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
If *info* = 0, *scond* contains the ratio of the smallest *s*(i) to the largest *s*(i).

amax — REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
Absolute value of the largest element of the matrix *A*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, the *i*th diagonal element of *A* is nonpositive.

Application Notes

If *scond* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *s*.

If *amax* is very close to overflow or very close to underflow, the matrix *A* should be scaled.

?pbequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite band matrix and reduce its condition number.

Syntax

```
call spbequ (uplo, n, kd, ab, ldab, s, scond, amax, info)
call dpbequ (uplo, n, kd, ab, ldab, s, scond, amax, info)
```

```
call cpbequ (uplo, n, kd, ab, ldab, s, scond, amax, info)
call zpbequ (uplo, n, kd, ab, ldab, s, scond, amax, info)
```

Description

This routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite matrix *A* in packed storage and reduce its condition number (with respect to the two-norm). The output array *s* returns scale factors computed as

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled matrix B with elements $b_{ij}=s(i)*a_{ij}*s(j)$ has diagonal elements equal to 1.
This choice of *s* puts the condition number of *B* within a factor *n* of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates whether the upper or lower triangular part of *A* is packed in the array *ab*:
If *uplo* = `'U'`, the array *ab* stores the upper triangular part of the matrix *A*.
If *uplo* = `'L'`, the array *ab* stores the lower triangular part of the matrix *A*.

n — `INTEGER`. The order of matrix *A* ($n \ge 0$).

kd — `INTEGER`. The number of super-diagonals or sub-diagonals in the matrix *A* ($kd \ge 0$).

ab — `REAL` for `spbequ`
`DOUBLE PRECISION` for `dpbequ`
`COMPLEX` for `cpbequ`
`DOUBLE COMPLEX` for `zpbequ`.
Array, `DIMENSION` (*ldab*,*).
The array *ap* contains either the upper or the lower triangular part of the matrix *A* (as specified by *uplo*) in *band storage* (see Matrix Storage Schemes).
The second dimension of *ab* must be at least max(1, *n*).

ldab — `INTEGER`. The leading dimension of the array *ab*.
($ldab \ge kd + 1$).

Output Parameters

s — REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
Array, DIMENSION (*n*).
If *info* = 0, the array *s* contains the scale factors for *A*.

scond — REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
If *info* = 0, *scond* contains the ratio of the smallest *s*(i) to the largest *s*(i).

amax — REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
Absolute value of the largest element of the matrix *A*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, the *i*th diagonal element of *A* is nonpositive.

Application Notes

If *scond* $\geq$ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *s*.

If *amax* is very close to overflow or very close to underflow, the matrix *A* should be scaled.

Driver Routines

Table 3-3 lists the LAPACK driver routines for solving systems of linear equations with real or complex matrices.

Table 3-3 Driver Routines for Solving Systems of Linear Equations

Matrix type, storage scheme	Simple Driver	Expert Driver
general	?gesv	?gesvx
general band	?gbsv	?gbsvx
general tridiagonal	?gtsv	?gtsvx
symmetric/Hermitian positive-definite	?posv	?posvx
symmetric/Hermitian positive-definite, packed storage	?ppsv	?ppsvx
symmetric/Hermitian positive-definite, band	?pbsv	?pbsvx
symmetric/Hermitian positive-definite, tridiagonal	?ptsv	?ptsvx
symmetric/Hermitian indefinite	?sysv /?hesv	?sysvx /?hesvx
symmetric/Hermitian indefinite, packed storage	?spsv /?hpsv	?spsvx /?hpsvx
complex symmetric	?sysv	?sysvx
complex symmetric, packed storage	?spsv	?spsvx

In this table **?** stands for **s** (single precision real), **d** (double precision real), **c** (single precision complex), or **z** (double precision complex).

?gesv

Computes the solution to the system of linear equations with a square matrix A and multiple right-hand sides.

Syntax

```
call sgesv (n, nrhs, a, lda, ipiv, b, ldb, info)
call dgesv (n, nrhs, a, lda, ipiv, b, ldb, info)
call cgesv (n, nrhs, a, lda, ipiv, b, ldb, info)
call zgesv (n, nrhs, a, lda, ipiv, b, ldb, info)
```

Description

This routine solves for X the system of linear equations $AX = B$, where A is an n-by-n matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The LU decomposition with partial pivoting and row interchanges is used to factor A as $A = P L U$, where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. The factored form of A is then used to solve the system of equations $AX = B$.

Input Parameters

n — `INTEGER`. The order of A; the number of rows in B ($n \geq 0$).

nrhs — `INTEGER`. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).

a, b — `REAL` for `sgesv`
`DOUBLE PRECISION` for `dgesv`
`COMPLEX` for `cgesv`
`DOUBLE COMPLEX` for `zgesv`.
Arrays: `a(lda,*)`, `b(ldb,*)`.

The array *a* contains the matrix A.
The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations.
The second dimension of *a* must be at least max(1,*n*), the second dimension of *b* at least max(1,*nrhs*).

lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

a	Overwritten by the factors *L* and *U* from the factorization of *A* = *P L U*; the unit diagonal elements of *L* are not stored .
b	Overwritten by the solution matrix *X*.
ipiv	INTEGER. Array, DIMENSION at least max(1,*n*). The pivot indices that define the permutation matrix *P*; row i of the matrix was interchanged with row *ipiv*(i).
info	INTEGER. If *info*=0, the execution is successful. If *info* = *-i*, the *i*th parameter had an illegal value. If *info* = *i*, *U*(*i*,*i*) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution could not be computed.

?gesvx

Computes the solution to the system of linear equations with a square matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

```
call sgesvx (fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c,
    b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info)
call dgesvx (fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c,
    b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info)
call cgesvx (fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c,
    b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info)
call zgesvx (fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c,
    b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info)
```

Description

This routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations $AX = B$, where A is an *n*-by-*n* matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?gesvx` performs the following steps:

1. If *fact* = 'E', real scaling factors *r* and *c* are computed to equilibrate the system:

trans = 'N': $\mathrm{diag}(r)*A*\mathrm{diag}(c) *\mathrm{diag}(c)^{-1}*X = \mathrm{diag}(r)*B$

trans = 'T': $(\mathrm{diag}(r)*A*\mathrm{diag}(c))^{T} *\mathrm{diag}(r)^{-1}*X = \mathrm{diag}(c)*B$

trans = 'C': $(\mathrm{diag}(r)*A*\mathrm{diag}(c))^{H} *\mathrm{diag}(r)^{-1}*X = \mathrm{diag}(c)*B$

Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by diag(*r*)**A**diag(*c*) and *B* by diag(*r*)**B* (if *trans*='N') or diag(*c*)**B* (if *trans* = 'T' or 'C').

2. If *fact* = 'N' or 'E', the *LU* decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as $A = P L U$, where *P* is a permutation matrix, *L* is a unit lower triangular matrix, and *U* is upper triangular.

3. If some $U_{i,i} = 0$, so that *U* is exactly singular, then the routine returns with *info* = i. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, *info* = $n + 1$ is returned as a warning, but the routine still goes on to solve for *X* and compute error bounds as described below.

4. The system of equations is solved for *X* using the factored form of *A*.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix *X* is premultiplied by diag(*c*) (if *trans* = 'N') or diag(*r*) (if *trans* = 'T' or 'C') so that it solves the original system before equilibration.

Input Parameters

fact — `CHARACTER*1`. Must be `'F'`, `'N'`, or `'E'`.

Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.

If *fact* = 'F': on entry, *af* and *ipiv* contain the factored form of *A*. If *equed* is not 'N', the matrix *A* has been equilibrated with scaling factors given by *r* and *c*.
a, *af*, and *ipiv* are not modified.

If *fact* = 'N', the matrix *A* will be copied to *af* and factored.
If *fact* = 'E', the matrix *A* will be equilibrated if necessary, then copied to *af* and factored.

trans CHARACTER*1. Must be 'N', 'T', or 'C'.

Specifies the form of the system of equations:

If *trans* = 'N', the system has the form $A X = B$ (No transpose);
If *trans* = 'T', the system has the form $A^T X = B$ (Transpose);
If *trans* = 'C', the system has the form $A^H X = B$ (Conjugate transpose);

n INTEGER. The number of linear equations; the order of the matrix *A* ($n \ge 0$).

nrhs INTEGER. The number of right hand sides; the number of columns of the matrices *B* and *X* ($nrhs \ge 0$).

a,af,b,work REAL for sgesvx
DOUBLE PRECISION for dgesvx
COMPLEX for cgesvx
DOUBLE COMPLEX for zgesvx.
Arrays: *a*(*lda*,*), *af*(*ldaf*,*), *b*(*ldb*,*), *work*(*).

The array *a* contains the matrix *A*. If *fact* = 'F' and *equed* is not 'N', then *A* must have been equilibrated by the scaling factors in *r* and/or *c*. The second dimension of *a* must be at least max(1,*n*).
The array *af* is an input argument if *fact* = 'F'. It contains the factored form of the matrix *A*, i.e., the factors *L* and *U* from the factorization $A = P L U$ as computed by ?getrf. If *equed* is not 'N', then *af* is the factored form of the equilibrated matrix *A*. The second dimension of *af* must be at least max(1,*n*).
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

work(*) is a workspace array.
The dimension of *work* must be at least max(1,4**n*) for real flavors, and at least max(1,2**n*) for complex flavors.

lda INTEGER. The first dimension of *a*; $lda \ge \max(1, n)$.

ldaf	INTEGER. The first dimension of *af*; *ldaf* ≥ max(1, *n*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).
ipiv	INTEGER. Array, DIMENSION at least max(1,*n*). The array *ipiv* is an input argument if *fact* = 'F'. It contains the pivot indices from the factorization *A* = *P L U* as computed by ?getrf; row i of the matrix was interchanged with row *ipiv*(i).
equed	CHARACTER*1. Must be 'N', 'R', 'C', or 'B'. *equed* is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done: If *equed* = 'N', no equilibration was done (always true if *fact* = 'N'); If *equed* = 'R', row equilibration was done and *A* has been premultiplied by diag(*r*); If *equed* = 'C', column equilibration was done and *A* has been postmultiplied by diag(*c*); If *equed* = 'B', both row and column equilibration was done; *A* has been replaced by diag(*r*)**A**diag(*c*).
r, c	REAL for single precision flavors; DOUBLE PRECISION for double precision flavors. Arrays: *r*(*n*), *c*(*n*). The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*. These arrays are input arguments if *fact* = 'F' only; otherwise they are output arguments. If *equed* = 'R' or 'B', *A* is multiplied on the left by diag(*r*); if *equed* = 'N' or 'C', *r* is not accessed. If *fact* = 'F' and *equed* = 'R' or 'B', each element of *r* must be positive. If *equed* = 'C' or 'B', *A* is multiplied on the right by diag(*c*); if *equed* = 'N' or 'R', *c* is not accessed. If *fact* = 'F' and *equed* = 'C' or 'B', each element of *c* must be positive.
ldx	INTEGER. The first dimension of the output array *x*; *ldx* ≥ max(1, *n*).
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*); used in real flavors only.

rwork — REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
Workspace array, DIMENSION at least max(1, 2*n); used in complex flavors only.

Output Parameters

x — REAL for sgesvx
DOUBLE PRECISION for dgesvx
COMPLEX for cgesvx
DOUBLE COMPLEX for zgesvx.
Array, DIMENSION (*ldx*,*).

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the *original* system of equations. Note that *A* and *B* are modified on exit if *equed* ≠ 'N', and the solution to the *equilibrated* system is:
diag(*c*)$^{-1}$*X, if *trans* = 'N' and *equed* = 'C' or 'B'; diag(*r*)$^{-1}$*X, if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'.
The second dimension of *x* must be at least max(1,*nrhs*).

a — Array *a* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'.
If *equed* ≠ 'N', *A* is scaled on exit as follows:
equed = 'R': *A* = diag(*r*)**A*
equed = 'C': *A* = *A**diag(*c*)
equed = 'B': *A* = diag(*r*)**A**diag(*c*)

af — If *fact* = 'N' or 'E', then *af* is an output argument and on exit returns the factors *L* and *U* from the factorization *A* = *P L U* of the original matrix *A*(if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *a* for the form of the equilibrated matrix.

b — Overwritten by diag(*r*)* *B* if *trans* = 'N' and *equed* = 'R' or 'B';
overwritten by diag(*c*)**B* if *trans* = 'T' and *equed* = 'C' or 'B';
not changed if *equed* = 'N'.

r, c — These arrays are output arguments if *fact* ≠ 'F'.
See the description of *r, c* in *Input Arguments* section.

rcond — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). The routine sets *rcond* =0 if the estimate underflows;

in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0,
for the working precision, the matrix may be poorly conditioned or even singular.

ferr, berr — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(1,*nrhs*). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

ipiv — If *fact* = 'N' or 'E', then *ipiv* is an output argument and on exit contains the pivot indices from the factorization $A = P L U$ of the original matrix *A*(if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E').

equed — If *fact* ≠ 'F' , then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

work, rwork — On exit, *work*(1) for real flavors, or *rwork*(1) for complex flavors, contains the reciprocal pivot growth factor norm(*A*)/norm(*U*). The "max absolute element" norm is used. If *work*(1) for real flavors, or *rwork*(1) for complex flavors is much less than 1, then the stability of the *LU* factorization of the (equilibrated) matrix *A* could be poor. This also means that the solution *x*, condition estimator *rcond*, and forward error bound *ferr* could be unreliable. If factorization fails with
$0 < info \le n$, then *work*(1) for real flavors, or *rwork*(1) for complex flavors contains the reciprocal pivot growth factor for the leading *info* columns of *A*.

info — INTEGER. If *info*=0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.
If *info* = *i*, and $i \le n$, then *U*(*i*,*i*) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.
If *info* = *i*, and $i = n + 1$, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

?gbsv

Computes the solution to the system of linear equations with a band matrix A and multiple right-hand sides.

Syntax

```
call sgbsv (n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info)
call dgbsv (n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info)
call cgbsv (n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info)
call zgbsv (n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info)
```

Description

This routine solves for X the real or complex system of linear equations $AX = B$, where A is an *n*-by-*n* band matrix with *kl* subdiagonals and *ku* superdiagonals, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The LU decomposition with partial pivoting and row interchanges is used to factor A as $A = L U$, where L is a product of permutation and unit lower triangular matrices with *kl* subdiagonals, and U is upper triangular with *kl*+*ku* superdiagonals. The factored form of A is then used to solve the system of equations $AX = B$.

Input Parameters

n	`INTEGER`. The order of A; the number of rows in B ($n \geq 0$).
kl	`INTEGER`. The number of sub-diagonals within the band of A ($kl \geq 0$).
ku	`INTEGER`. The number of super-diagonals within the band of A ($ku \geq 0$).
nrhs	`INTEGER`. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
ab, b	`REAL` for `sgbsv` `DOUBLE PRECISION` for `dgbsv` `COMPLEX` for `cgbsv` `DOUBLE COMPLEX` for `zgbsv`. Arrays: *ab*(*ldab*,*), *b*(*ldb*,*). The array *ab* contains the matrix A in band storage (see Matrix Storage Schemes).

The second dimension of *ab* must be at least max(1, *n*).
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.
The second dimension of *b* must be at least max(1,*nrhs*).

ldab — INTEGER. The first dimension of the array *ab*.
($ldab \geq 2kl + ku + 1$)

ldb — INTEGER. The first dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

ab — Overwritten by *L* and *U*. The diagonal and $kl + ku$ super-diagonals of *U* are stored in the first $1 + kl + ku$ rows of *ab*. The multipliers used to form *L* are stored in the next *kl* rows.

b — Overwritten by the solution matrix *X*.

ipiv — INTEGER.
Array, DIMENSION at least max(1,*n*).
The pivot indices: row *i* was interchanged with row *ipiv*(*i*).

info — INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, *U*(*i*,*i*) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution could not be computed.

?gbsvx

Computes the solution to the real or complex system of linear equations with a band matrix A and multiple right-hand sides, and provides error bounds on the solution.

```
call sgbsvx (fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv,
    equed, r, c, b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info)
call dgbsvx (fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv,
    equed, r, c, b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info)
call cgbsvx (fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv,
    equed, r, c, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info)
```

```
call zgbsvx (fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv,
    equed, r, c, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info)
```

Description

This routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations $AX = B$, $A^T X = B$, or $A^H X = B$, where A is a band matrix of order *n* with *kl* subdiagonals and *ku* superdiagonals, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?gbsvx` performs the following steps:

1. If *fact* = 'E', real scaling factors *r* and *c* are computed to equilibrate the system:

trans = 'N': $\text{diag}(r)*A*\text{diag}(c) *\text{diag}(c)^{-1}*X = \text{diag}(r)*B$

trans = 'T': $(\text{diag}(r)*A*\text{diag}(c))^T *\text{diag}(r)^{-1}*X = \text{diag}(c)*B$

trans = 'C': $(\text{diag}(r)*A*\text{diag}(c))^H *\text{diag}(r)^{-1}*X = \text{diag}(c)*B$

Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by diag(*r*)**A**diag(*c*) and *B* by diag(*r*)**B* (if *trans*='N') or diag(*c*)**B* (if *trans* = 'T' or 'C').

2. If *fact* = 'N' or 'E', the *LU* decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as $A = L U$, where *L* is a product of permutation and unit lower triangular matrices with *kl* subdiagonals, and *U* is upper triangular with *kl*+*ku* superdiagonals.

3. If some $U_{i,i} = 0$, so that *U* is exactly singular, then the routine returns with *info* = i. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, *info* = *n* + 1 is returned as a warning, but the routine still goes on to solve for *X* and compute error bounds as described below.

4. The system of equations is solved for *X* using the factored form of *A*.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix *X* is premultiplied by diag(*c*) (if *trans* = 'N') or diag(*r*) (if *trans* = 'T' or 'C') so that it solves the original system before equilibration.

Input Parameters

fact	CHARACTER*1. Must be 'F', 'N', or 'E'. Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored. If *fact* = 'F': on entry, *afb* and *ipiv* contain the factored form of *A*. If *equed* is not 'N', the matrix *A* has been equilibrated with scaling factors given by *r* and *c*. *ab*, *afb*, and *ipiv* are not modified. If *fact* = 'N', the matrix *A* will be copied to *afb* and factored. If *fact* = 'E', the matrix *A* will be equilibrated if necessary, then copied to *afb* and factored.
trans	CHARACTER*1. Must be 'N', 'T', or 'C'. Specifies the form of the system of equations: If *trans* = 'N', the system has the form $A X = B$ (No transpose); If *trans* = 'T', the system has the form $A^T X = B$ (Transpose); If *trans* = 'C', the system has the form $A^H X = B$ (Conjugate transpose);
n	INTEGER. The number of linear equations; the order of the matrix *A* ($n \geq 0$).
kl	INTEGER. The number of sub-diagonals within the band of *A* ($kl \geq 0$).
ku	INTEGER. The number of super-diagonals within the band of *A* ($ku \geq 0$).
nrhs	INTEGER. The number of right hand sides; the number of columns of the matrices *B* and *X* ($nrhs \geq 0$).
ab,afb,b,work	REAL for sgesvx DOUBLE PRECISION for dgesvx COMPLEX for cgesvx DOUBLE COMPLEX for zgesvx. Arrays: *a*(*lda*,*), *af*(*ldaf*,*), *b*(*ldb*,*), *work*(*). The array *ab* contains the matrix *A* in band storage (see Matrix Storage Schemes). The second dimension of *ab* must be at least max(1, *n*). If *fact* = 'F' and *equed* is not 'N', then *A* must have been equilibrated by the scaling factors in *r* and/or *c*. The array *afb* is an input argument if *fact* = 'F' . The second dimension of *afb* must be at least max(1,*n*). It contains the factored form of the matrix *A*, i.e., the factors *L* and *U* from the

factorization $A = L U$ as computed by ?gbtrf. U is stored as an upper triangular band matrix with *kl* + *ku* super-diagonals in the first 1 + *kl* + *ku* rows of *afb*. The multipliers used during the factorization are stored in the next *kl* rows.
If *equed* is not 'N', then *afb* is the factored form of the equilibrated matrix *A*.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

work(*) is a workspace array.
The dimension of *work* must be at least max(1,3**n*) for real flavors, and at least max(1,2**n*) for complex flavors.

ldab INTEGER. The first dimension of *ab*; *ldab* ≥ *kl*+*ku*+1.

ldafb INTEGER. The first dimension of *afb*;
ldafb ≥ 2**kl*+*ku*+1.

ldb INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

ipiv INTEGER.
Array, DIMENSION at least max(1,*n*).
The array *ipiv* is an input argument if *fact* = 'F'.
It contains the pivot indices from the factorization
$A = L U$ as computed by ?gbtrf; row i of the matrix was interchanged with row *ipiv*(i).

equed CHARACTER*1. Must be 'N', 'R', 'C', or 'B'.
equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:
If *equed* = 'N', no equilibration was done (always true if *fact* = 'N');
If *equed* = 'R', row equilibration was done and *A* has been premultiplied by diag(*r*);
If *equed* = 'C', column equilibration was done and *A* has been postmultiplied by diag(*c*);
If *equed* = 'B', both row and column equilibration was done; *A* has been replaced by diag(*r*)**A**diag(*c*).

r, c REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
Arrays: *r*(*n*), *c*(*n*).
The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*. These arrays are input arguments if *fact* = 'F' only;

otherwise they are output arguments.
If *equed* = 'R' or 'B', *A* is multiplied on the left by diag(*r*); if *equed* = 'N' or 'C', *r* is not accessed.
If *fact* = 'F' and *equed* = 'R' or 'B', each element of *r* must be positive.
If *equed* = 'C' or 'B', *A* is multiplied on the right by diag(*c*); if *equed* = 'N' or 'R', *c* is not accessed.
If *fact* = 'F' and *equed* = 'C' or 'B', each element of *c* must be positive.

ldx INTEGER. The first dimension of the output array *x*; *ldx* ≥ max(1, *n*).

iwork INTEGER.
Workspace array, DIMENSION at least max(1, *n*); used in real flavors only.

rwork REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
Workspace array, DIMENSION at least max(1, *n*); used in complex flavors only.

Output Parameters

x REAL for sgbsvx
DOUBLE PRECISION for dgbsvx
COMPLEX for cgbsvx
DOUBLE COMPLEX for zgbsvx.
Array, DIMENSION (*ldx*, *).

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the *original* system of equations. Note that *A* and *B* are modified on exit if *equed* ≠ 'N', and the solution to the *equilibrated* system is:
$\text{diag}(c)^{-1} * X$, if *trans* = 'N' and *equed* = 'C' or 'B'; $\text{diag}(r)^{-1} * X$, if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'.
The second dimension of *x* must be at least max(1,*nrhs*).

ab Array *ab* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'.
If *equed* ≠ 'N', *A* is scaled on exit as follows:
equed = 'R': $A = \text{diag}(r) * A$
equed = 'C': $A = A * \text{diag}(c)$
equed = 'B': $A = \text{diag}(r) * A * \text{diag}(c)$

afb If *fact* = 'N' or 'E', then *afb* is an output argument and on exit returns details of the *LU* factorization of the original matrix *A*(if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *ab* for the form of the equilibrated matrix.

b — Overwritten by diag(*r*)**b* if *trans* = 'N' and *equed* = 'R' or 'B'; overwritten by diag(*c*)**b* if *trans* = 'T' and *equed* = 'C' or 'B'; not changed if *equed* = 'N'.

r, c — These arrays are output arguments if *fact* ≠ 'F'. See the description of *r, c* in *Input Arguments* section.

rcond — REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr, berr — REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1,*nrhs*). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

ipiv — If *fact* = 'N' or 'E', then *ipiv* is an output argument and on exit contains the pivot indices from the factorization *A* = *L U* of the original matrix *A*(if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E').

equed — If *fact* ≠ 'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

work, rwork — On exit, *work*(1) for real flavors, or *rwork*(1) for complex flavors, contains the reciprocal pivot growth factor norm(*A*)/norm(*U*). The "max absolute element" norm is used. If *work*(1) for real flavors, or *rwork*(1) for complex flavors is much less than 1, then the stability of the *LU* factorization of the (equilibrated) matrix *A* could be poor. This also means that the solution *x*, condition estimator *rcond*, and forward error bound *ferr* could be unreliable. If factorization fails with 0 < *info* ≤ *n*, then *work*(1) for real flavors, or *rwork*(1) for complex flavors contains the reciprocal pivot growth factor for the leading *info* columns of *A*.

info — INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i*, and *i* ≤ *n*, then *U*(*i*,*i*) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned. If *info* = *i*, and *i* = *n* +1, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

?gtsv

Computes the solution to the system of linear equations with a tridiagonal matrix A and multiple right-hand sides.

Syntax

```
call sgtsv (n, nrhs, dl, d, du, b, ldb, info)
call dgtsv (n, nrhs, dl, d, du, b, ldb, info)
call cgtsv (n, nrhs, dl, d, du, b, ldb, info)
call zgtsv (n, nrhs, dl, d, du, b, ldb, info)
```

Description

This routine solves for X the system of linear equations $AX = B$, where A is an n-by-n tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.
The routine uses Gaussian elimination with partial pivoting.

Note that the equation $A^T X = B$ may be solved by interchanging the order of the arguments *du* and *dl*.

Input Parameters

n — `INTEGER`. The order of A; the number of rows in B ($n \geq 0$).

nrhs — `INTEGER`. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).

dl, d, du, b — `REAL` for `sgtsv`
`DOUBLE PRECISION` for `dgtsv`
`COMPLEX` for `cgtsv`
`DOUBLE COMPLEX` for `zgtsv`.

Arrays: *dl*(*n* - 1), *d*(*n*), *du*(*n* - 1), *b*(*ldb*, *).
The array *dl* contains the (*n* - 1) subdiagonal elements of *A*.
The array *d* contains the diagonal elements of *A*.
The array *du* contains the (*n* - 1) superdiagonal elements of *A*.
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.
The second dimension of *b* must be at least max(1,*nrhs*).

ldb — INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

dl — Overwritten by the (*n*-2) elements of the second superdiagonal of the upper triangular matrix *U* from the *LU* factorization of A. These elements are stored in *dl*(1), ... , *dl*(*n*-2).

d — Overwritten by the *n* diagonal elements of *U*.

du — Overwritten by the (*n*-1) elements of the first superdiagonal of *U*.

b — Overwritten by the solution matrix *X*.

info — INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, *U*(*i*,*i*) is exactly zero, and the solution has not been computed. The factorization has not been completed unless *i* = *n*.

?gtsvx

Computes the solution to the real or complex system of linear equations with a tridiagonal matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

```
call sgtsvx (fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b,
    ldb, x, ldx, rcond, ferr, berr, work, iwork, info)
call dgtsvx (fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b,
    ldb, x, ldx, rcond, ferr, berr, work, iwork, info)
call cgtsvx (fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b,
    ldb, x, ldx, rcond, ferr, berr, work, rwork, info)
```

```
call zgtsvx (fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b,
    ldb, x, ldx, rcond, ferr, berr, work, rwork, info)
```

Description

This routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations $AX = B$, $A^TX = B$, or $A^HX = B$, where A is a tridiagonal matrix of order *n*, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?gtsvx` performs the following steps:

1. If *fact* = 'N', the *LU* decomposition is used to factor the matrix *A* as $A = LU$, where *L* is a product of permutation and unit lower bidiagonal matrices and *U* is an upper triangular matrix with nonzeroes in only the main diagonal and first two superdiagonals.

2. If some $U_{i,i} = 0$, so that *U* is exactly singular, then the routine returns with *info* = `i`. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, *info* = *n* + 1 is returned as a warning, but the routine still goes on to solve for *X* and compute error bounds as described below.

3. The system of equations is solved for *X* using the factored form of *A*.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact — `CHARACTER*1`. Must be `'F'` or `'N'`.

Specifies whether or not the factored form of the matrix *A* has been supplied on entry.

If *fact* = `'F'`: on entry, *dlf*, *df*, *duf*, *du2*, and *ipiv* contain the factored form of *A*; arrays *dl*, *d*, *du*, *dlf*, *df*, *duf*, *du2*, and *ipiv* will not be modified.

If *fact* = `'N'`, the matrix *A* will be copied to *dlf*, *df*, and *duf* and factored.

trans — `CHARACTER*1`. Must be `'N'`, `'T'`, or `'C'`.

Specifies the form of the system of equations:

If `trans` = `'N'`, the system has the form $A X = B$ (No transpose);
If `trans` = `'T'`, the system has the form $A^T X = B$ (Transpose);
If `trans` = `'C'`, the system has the form $A^H X = B$ (Conjugate transpose);

n — `INTEGER`. The number of linear equations; the order of the matrix *A* ($n \geq 0$).

nrhs — `INTEGER`. The number of right hand sides; the number of columns of the matrices *B* and *X* ($nrhs \geq 0$).

dl,d,du,dlf,df,
duf,du2,b,x,work — `REAL` for `sgtsvx`
`DOUBLE PRECISION` for `dgtsvx`
`COMPLEX` for `cgtsvx`
`DOUBLE COMPLEX` for `zgtsvx`.
Arrays:

dl, dimension (*n* - 1), contains the subdiagonal elements of *A*.

d, dimension (*n*), contains the diagonal elements of *A*.

du, dimension (*n* - 1), contains the superdiagonal elements of *A*.

dlf, dimension (*n* - 1). If *fact* = 'F' , then *dlf* is an input argument and on entry contains the (*n* - 1) multipliers that define the matrix *L* from the *LU* factorization of *A* as computed by `?gttrf`.

df, dimension (*n*). If *fact* = 'F' , then *df* is an input argument and on entry contains the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.

duf, dimension (*n* - 1). If *fact* = 'F' , then *duf* is an input argument and on entry contains the (*n* - 1) elements of the first super-diagonal of *U*.

du2, dimension (*n* - 2). If *fact* = 'F' , then *du2* is an input argument and on entry contains the (*n* - 2) elements of the second super-diagonal of *U*.

b(*ldb*, *) contains the right-hand side matrix *B*. The second dimension of *b* must be at least max(1,*nrhs*).

x(*ldx*, *) contains the solution matrix *X*. The second dimension of *x* must be at least max(1,*nrhs*).

work (*) is a workspace array;
the dimension of *work* must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.

ldb — `INTEGER`. The first dimension of *b*; $ldb \geq \max(1, n)$.

ldx — `INTEGER`. The first dimension of *x*; $ldx \geq \max(1, n)$.

ipiv — INTEGER.
Array, DIMENSION at least max(1,*n*). If *fact* = 'F' , then *ipiv* is an input argument and on entry contains the pivot indices, as returned by ?gttrf.

iwork — INTEGER.
Workspace array, DIMENSION (*n*). Used for real flavors only.

rwork — REAL for cgtsvx
DOUBLE PRECISION for zgtsvx.
Workspace array, DIMENSION (*n*). Used for complex flavors only.

Output Parameters

x — REAL for sgtsvx
DOUBLE PRECISION for dgtsvx
COMPLEX for cgtsvx
DOUBLE COMPLEX for zgtsvx.
Array, DIMENSION (*ldx*,*).

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X*. The second dimension of *x* must be at least max(1,*nrhs*).

dlf — If *fact* = 'N' , then *dlf* is an output argument and on exit contains the (*n* - 1) multipliers that define the matrix *L* from the *LU* factorization of *A*.

df — If *fact* = 'N' , then *df* is an output argument and on exit contains the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.

duf — If *fact* = 'N' , then *duf* is an output argument and on exit contains the (*n* - 1) elements of the first super-diagonal of *U*.

du2 — If *fact* = 'N' , then *du2* is an output argument and on exit contains the (*n* - 2) elements of the second super-diagonal of *U*.

ipiv — The array *ipiv* is an output argument if *fact* = 'N' and, on exit, contains the pivot indices from the factorization
A = *L U*; row *i* of the matrix was interchanged with row *ipiv*(*i*). The value of *ipiv*(*i*) will always be either *i* or *i*+1; *ipiv*(*i*)=*i* indicates a row interchange was not required.

rcond — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal condition number of the matrix *A*.

If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr, berr — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(1,*nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.

info — INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, and $i \le n$, then $U(i,i)$ is exactly zero. The factorization has not been completed unless $i = n$, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.
If *info* = *i*, and $i = n+1$, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

?posv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive definite matrix A and multiple right-hand sides.

Syntax

```
call sposv (uplo, n, nrhs, a, lda, b, ldb, info)
call dposv (uplo, n, nrhs, a, lda, b, ldb, info)
call cposv (uplo, n, nrhs, a, lda, b, ldb, info)
call zposv (uplo, n, nrhs, a, lda, b, ldb, info)
```

Description

This routine solves for *X* the real or complex system of linear equations $AX = B$, where A is an *n*-by-*n* symmetric/Hermitian positive definite matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The Cholesky decomposition is used to factor A as $A = U^H U$ if *uplo* = 'U' or $A = LL^H$ if `uplo` = `'L'`, where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations $AX = B$.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If *uplo* = `'U'`, the array *a* stores the upper triangular part of the matrix A, and A is factored as $U^H U$. If *uplo* = `'L'`, the array *a* stores the lower triangular part of the matrix A; A is factored as LL^H.
n	`INTEGER`. The order of matrix A ($n \ge 0$).
nrhs	`INTEGER`. The number of right-hand sides; the number of columns in B ($nrhs \ge 0$).
a, b	`REAL` for `sposv` `DOUBLE PRECISION` for `dposv` `COMPLEX` for `cposv` `DOUBLE COMPLEX` for `zposv`. Arrays: *a*(*lda*,*), *b*(*ldb*,*). The array *a* contains either the upper or the lower triangular part of the matrix A (see *uplo*). The second dimension of *a* must be at least max(1, *n*). The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).
lda	`INTEGER`. The first dimension of *a*; $lda \ge \max(1, n)$.
ldb	`INTEGER`. The first dimension of *b*; $ldb \ge \max(1, n)$.

Output Parameters

a	If *info*=0, the upper or lower triangular part of *a* is overwritten by the Cholesky factor U or L, as specified by *uplo*.
b	Overwritten by the solution matrix X.
info	`INTEGER`. If *info*=0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i*, the leading minor of order *i* (and hence the matrix A itself) is not positive definite, so the factorization could not be completed, and the solution has not been computed.

?posvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric or Hermitian positive definite matrix A, and provides error bounds on the solution.

Syntax

```
call sposvx (fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x,
    ldx, rcond, ferr, berr, work, iwork, info)
call dposvx (fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x,
    ldx, rcond, ferr, berr, work, iwork, info)
call cposvx (fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x,
    ldx, rcond, ferr, berr, work, rwork, info)
call zposvx (fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x,
    ldx, rcond, ferr, berr, work, rwork, info)
```

Description

This routine uses the Cholesky factorization $A=U^HU$ or $A=LL^H$ to compute the solution to a real or complex system of linear equations $AX=B$, where A is a n-by-n real symmetric/Hermitian positive definite matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?posvx` performs the following steps:

1. If `fact` = 'E', real scaling factors `s` are computed to equilibrate the system:

$$\text{diag}(s)*A*\text{diag}(s)*\text{diag}(s)^{-1}*X = \text{diag}(s)*B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A, but if equilibration is used, A is overwritten by diag(s)*A*diag(s) and B by diag(s)*B .

2. If `fact` = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if `fact` = 'E') as

$A = U^H U$, if *uplo* = 'U', or
$A = L L^H$, if *uplo* = 'L',
where U is an upper triangular matrix and L is a lower triangular matrix.

3. If the leading *i*-by-*i* principal minor is not positive definite, then the routine returns with *info* = *i*. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, *info* = *n* + 1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by diag(*s*) so that it solves the original system before equilibration.

Input Parameters

fact — CHARACTER*1. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.

If *fact* = 'F': on entry, *af* contains the factored form of A. If *equed* = 'Y', the matrix A has been equilibrated with scaling factors given by *s*.
a and *af* will not be modified.

If *fact* = 'N', the matrix A will be copied to *af* and factored.
If *fact* = 'E', the matrix A will be equilibrated if necessary, then copied to *af* and factored.

uplo — CHARACTER*1. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of A is stored and how A is factored:
If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix A, and A is factored as $U^H U$.
If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix A; A is factored as LL^H.

n — INTEGER. The order of matrix A ($n \geq 0$).

nrhs — INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).

a,af,b,work
REAL for sposvx
DOUBLE PRECISION for dposvx
COMPLEX for cposvx
DOUBLE COMPLEX for zposvx.
Arrays: *a*(*lda*,*), *af*(*ldaf*,*), *b*(*ldb*,*), *work*(*).

The array *a* contains the matrix *A* as specified by *uplo*. If *fact* = 'F' and *equed* = 'Y', then *A* must have been equilibrated by the scaling factors in *s*, and *a* must contain the equilibrated matrix diag(*s*)**A**diag(*s*). The second dimension of *a* must be at least max(1,*n*).

The array *af* is an input argument if *fact* = 'F' .
It contains the triangular factor *U* or *L* from the Cholesky factorization of *A* in the same storage format as *A*. If *equed* is not 'N', then *af* is the factored form of the equilibrated matrix diag(*s*)**A**diag(*s*). The second dimension of *af* must be at least max(1,*n*).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

work(*) is a workspace array.
The dimension of *work* must be at least max(1,3**n*) for real flavors, and at least max(1,2**n*) for complex flavors.

lda
INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).

ldaf
INTEGER. The first dimension of *af*; *ldaf* ≥ max(1, *n*).

ldb
INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

equed
CHARACTER*1. Must be 'N' or 'Y'.
equed is an input argument if *fact* = 'F' . It specifies the form of equilibration that was done:
If *equed* = 'N', no equilibration was done (always true if *fact* = 'N');
If *equed* = 'Y', equilibration was done and *A* has been replaced by diag(*s*)**A**diag(*s*).

s
REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
Array, DIMENSION (*n*).
The array *s* contains the scale factors for *A*. This array is an input argument if

	fact = 'F' only; otherwise it is an output argument. If *equed* = 'N' , *s* is not accessed. If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.
ldx	INTEGER. The first dimension of the output array *x*; *ldx* ≥ max(1, *n*).
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*); used in real flavors only.
rwork	REAL for cposvx; DOUBLE PRECISION for zposvx. Workspace array, DIMENSION at least max(1, *n*); used in complex flavors only.

Output Parameters

x	REAL for sposvx DOUBLE PRECISION for dposvx COMPLEX for cposvx DOUBLE COMPLEX for zposvx. Array, DIMENSION (*ldx*,*). If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the *original* system of equations. Note that if *equed* = 'Y', *A* and *B* are modified on exit, and the solution to the equilibrated system is diag(*s*)$^{-1}$***X*. The second dimension of *x* must be at least max(1,*nrhs*).
a	Array *a* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'. If *fact* = 'E' and *equed* = 'Y', *A* is overwritten by diag(*s*)**A**diag(*s*)
af	If *fact* = 'N' or 'E', then *af* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A=U^HU$ or $A=LL^H$ of the original matrix *A*(if *fact* = 'N'), or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *a* for the form of the equilibrated matrix.
b	Overwritten by diag(*s*)**B* , if *equed* = 'Y'; not changed if *equed* = 'N'.
s	This array is an output argument if *fact* ≠ 'F' . See the description of *s* in *Input Arguments* section.
rcond	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix *A* after

equilibration (if done). If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr, berr — `REAL` for single precision flavors.
`DOUBLE PRECISION` for double precision flavors.
Arrays, `DIMENSION` at least max(1,*nrhs*). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

equed — If *fact* ≠ 'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

info — `INTEGER`. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, and *i* ≤ *n*, the leading minor of order *i* (and hence the matrix *A* itself) is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed; *rcond* = 0 is returned.
If *info* = *i*, and *i* = *n* +1, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

?ppsv

Computes the solution to the system of linear equations with a symmetric (Hermitian) positive definite packed matrix A and multiple right-hand sides.

Syntax

```
call sppsv (uplo, n, nrhs, ap, b, ldb, info)
call dppsv (uplo, n, nrhs, ap, b, ldb, info)
call cppsv (uplo, n, nrhs, ap, b, ldb, info)
call zppsv (uplo, n, nrhs, ap, b, ldb, info)
```

Description

This routine solves for X the real or complex system of linear equations $AX = B$, where A is an *n*-by-*n* real symmetric/Hermitian positive definite matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The Cholesky decomposition is used to factor A as $A = U^H U$ if `uplo` = 'U'

or $A = LL^H$ if `uplo` = 'L', where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations $AX = B$.

Input Parameters

uplo	`CHARACTER*1`. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix A, and A is factored as $U^H U$. If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix A; A is factored as LL^H.
n	`INTEGER`. The order of matrix A ($n \geq 0$).
nrhs	`INTEGER`. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
ap, b	`REAL` for `sppsv` `DOUBLE PRECISION` for `dppsv` `COMPLEX` for `cppsv` `DOUBLE COMPLEX` for `zppsv`. Arrays: *ap*(*), *b*(*ldb*,*). The array *ap* contains either the upper or the lower triangular part of the matrix A (as specified by *uplo*) in *packed storage* (see Matrix Storage Schemes). The dimension of *ap* must be at least max(1,*n*(*n*+1)/2). The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).
ldb	`INTEGER`. The first dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

ap	If *info*=0, the upper or lower triangular part of A in packed storage is overwritten by the Cholesky factor U or L, as specified by *uplo*.

b Overwritten by the solution matrix *X*.

info `INTEGER`. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, the leading minor of order *i* (and hence the matrix *A* itself) is not positive definite, so the factorization could not be completed, and the solution has not been computed.

?ppsvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite packed matrix A, and provides error bounds on the solution.

Syntax

```
call sppsvx (fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx,
    rcond, ferr, berr, work, iwork, info)
call dppsvx (fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx,
    rcond, ferr, berr, work, iwork, info)
call cppsvx (fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx,
    rcond, ferr, berr, work, rwork, info)
call zppsvx (fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx,
    rcond, ferr, berr, work, rwork, info)
```

Description

This routine uses the Cholesky factorization $A=U^HU$ or $A=LL^H$ to compute the solution to a real or complex system of linear equations $AX = B$, where A is a *n*-by-*n* symmetric or Hermitian positive definite matrix stored in packed format, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?ppsvx` performs the following steps:

1. If *fact* = 'E', real scaling factors *s* are computed to equilibrate the system:

$$\text{diag}(s)*A*\text{diag}(s)*\text{diag}(s)^{-1}*X = \text{diag}(s)*B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by diag(`s`)*`A`*diag(`s`) and *B* by diag(`s`)*`B` .

2. If `fact` = 'N' or 'E', the Cholesky decomposition is used to factor the matrix *A* (after equilibration if `fact` = 'E') as

$A = U^H U$, if `uplo` = 'U', or
$A = L L^H$, if `uplo` = 'L',
where *U* is an upper triangular matrix and *L* is a lower triangular matrix.

3. If the leading `i`-by-`i` principal minor is not positive definite, then the routine returns with `info` = `i`. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, `info` = `n` + 1 is returned as a warning, but the routine still goes on to solve for *X* and compute error bounds as described below.

4. The system of equations is solved for *X* using the factored form of *A*.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix *X* is premultiplied by diag(`s`) so that it solves the original system before equilibration.

Input Parameters

`fact` — `CHARACTER*1`. Must be `'F'`, `'N'`, or `'E'`.

Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.

If `fact` = `'F'`: on entry, `afp` contains the factored form of *A*. If `equed` = `'Y'`, the matrix *A* has been equilibrated with scaling factors given by `s`.
`ap` and `afp` will not be modified.

If `fact` = `'N'`, the matrix *A* will be copied to `afp` and factored.
If `fact` = `'E'`, the matrix *A* will be equilibrated if necessary, then copied to `afp` and factored.

`uplo` — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates whether the upper or lower triangular part of *A* is stored and how *A* is factored:

If *uplo* = 'U', the array *ap* stores the upper triangular part of the matrix *A*, and *A* is factored as $U^H U$.
If *uplo* = 'L', the array *ap* stores the lower triangular part of the matrix *A*; *A* is factored as LL^H.

n — INTEGER. The order of matrix *A* ($n \geq 0$).

nrhs — INTEGER. The number of right-hand sides; the number of columns in *B* ($nrhs \geq 0$).

ap,afp,b,work — REAL for sppsvx
DOUBLE PRECISION for dppsvx
COMPLEX for cppsvx
DOUBLE COMPLEX for zppsvx.
Arrays: *ap*(*), *afp*(*), *b*(*ldb*,*), *work* (*).

The array *ap* contains the upper or lower triangle of the original symmetric/Hermitian matrix A in *packed storage* (see Matrix Storage Schemes). In case when *fact* = 'F' and *equed* = 'Y', *ap* must contain the equilibrated matrix diag(*s*)**A**diag(*s*).

The array *afp* is an input argument if *fact* = 'F' and contains the triangular factor *U* or *L* from the Cholesky factorization of *A* in the same storage format as *A*. If *equed* is not 'N', then *afp* is the factored form of the equilibrated matrix *A*.
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.
work (*) is a workspace array.

The dimension of arrays *ap* and *afp* must be at least max(1,*n*(*n*+1)/2); the second dimension of *b* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.

ldb — INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

equed — CHARACTER*1. Must be 'N' or 'Y'.
equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:
If *equed* = 'N', no equilibration was done (always true if *fact* = 'N');
If *equed* = 'Y', equilibration was done and *A* has been replaced by diag(*s*)**A**diag(*s*).

s	REAL for single precision flavors; DOUBLE PRECISION for double precision flavors. Array, DIMENSION (*n*). The array *s* contains the scale factors for *A*. This array is an input argument if *fact* = 'F' only; otherwise it is an output argument. If *equed* = 'N' , *s* is not accessed. If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.
ldx	INTEGER. The first dimension of the output array *x*; *ldx* ≥ max(1, *n*).
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*); used in real flavors only.
rwork	REAL for cppsvx; DOUBLE PRECISION for zppsvx. Workspace array, DIMENSION at least max(1, *n*); used in complex flavors only.

Output Parameters

x	REAL for sppsvx DOUBLE PRECISION for dppsvx COMPLEX for cppsvx DOUBLE COMPLEX for zppsvx. Array, DIMENSION (*ldx*,*). If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the *original* system of equations. Note that if *equed* = 'Y', *A* and *B* are modified on exit, and the solution to the equilibrated system is diag(s)$^{-1}$**X*. The second dimension of *x* must be at least max(1,*nrhs*).
ap	Array *ap* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'. If *fact* = 'E' and *equed* = 'Y', *A* is overwritten by diag(*s*)**A**diag(*s*)
afp	If *fact* = 'N' or 'E', then *afp* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A=U^HU$ or $A=LL^H$ of the original matrix *A*(if *fact* = 'N'), or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *ap* for the form of the equilibrated matrix.
b	Overwritten by diag(*s*)**B* , if *equed* = 'Y'; not changed if *equed* = 'N'.
s	This array is an output argument if *fact* ≠ 'F' . See the description of *s* in *Input Arguments* section.

rcond	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.
ferr, berr	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1,*nrhs*). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.
equed	If *fact* ≠ 'F' , then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i*, and *i* ≤ *n*, the leading minor of order *i* (and hence the matrix *A* itself) is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed; *rcond* = 0 is returned. If *info* = *i*, and *i* = *n* +1, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

?pbsv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive definite band matrix A and multiple right-hand sides.

Syntax

```
call spbsv (uplo, n, kd, nrhs, ab, ldab, b, ldb, info)
call dpbsv (uplo, n, kd, nrhs, ab, ldab, b, ldb, info)
call cpbsv (uplo, n, kd, nrhs, ab, ldab, b, ldb, info)
call zpbsv (uplo, n, kd, nrhs, ab, ldab, b, ldb, info)
```

Description

This routine solves for X the real or complex system of linear equations $AX = B$, where A is an *n*-by-*n* symmetric/Hermitian positive definite band matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The Cholesky decomposition is used to factor A as $A = U^H U$ if *uplo* = 'U'

or $A = LL^H$ if *uplo* = 'L', where U is an upper triangular band matrix and L is a lower triangular band matrix, with the same number of superdiagonals or subdiagonals as A. The factored form of A is then used to solve the system of equations $AX = B$.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored in the array *ab*, and how A is factored: If *uplo* = 'U', the array *ab* stores the upper triangular part of the matrix A, and A is factored as $U^H U$. If *uplo* = 'L', the array *ab* stores the lower triangular part of the matrix A; A is factored as LL^H.
n	INTEGER. The order of matrix A ($n \geq 0$).
kd	INTEGER. The number of superdiagonals of the matrix A if *uplo* = 'U', or the number of subdiagonals if *uplo* = 'L' ($kd \geq 0$).
nrhs	INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
ab, b	REAL for spbsv DOUBLE PRECISION for dpbsv COMPLEX for cpbsv DOUBLE COMPLEX for zpbsv. Arrays: *ab*(*ldab*, *), *b*(*ldb*, *). The array *ab* contains either the upper or the lower triangular part of the matrix A (as specified by *uplo*) in *band storage* (see Matrix Storage Schemes). The second dimension of *ab* must be at least max(1, *n*). The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).
ldab	INTEGER. The first dimension of the array *ab*. ($ldab \geq kd + 1$)
ldb	INTEGER. The first dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

ab	The upper or lower triangular part of A (in band storage) is overwritten by the Cholesky factor U or L, as specified by *uplo*, in the same storage format as A.
b	Overwritten by the solution matrix X.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i*, the leading minor of order *i* (and hence the matrix A itself) is not positive definite, so the factorization could not be completed, and the solution has not been computed.

?pbsvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite band matrix A, and provides error bounds on the solution.

Syntax

```
call spbsvx (fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b,
    ldb, x, ldx, rcond, ferr, berr, work, iwork, info)
call dpbsvx (fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b,
    ldb, x, ldx, rcond, ferr, berr, work, iwork, info)
call cpbsvx (fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b,
    ldb, x, ldx, rcond, ferr, berr, work, iwork, info)
call zpbsvx (fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b,
    ldb, x, ldx, rcond, ferr, berr, work, iwork, info)
```

Description

This routine uses the Cholesky factorization $A=U^HU$ or $A=LL^H$ to compute the solution to a real or complex system of linear equations $AX = B$, where A is a *n*-by-*n* symmetric or Hermitian positive definite band matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?pbsvx performs the following steps:

1. If *fact* = 'E', real scaling factors *s* are computed to equilibrate the system:

$$\text{diag}(s)*A*\text{diag}(s)*\text{diag}(s)^{-1}*X = \text{diag}(s)*B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by diag(*s*)**A**diag(*s*) and *B* by diag(*s*)**B* .

2. If *fact* = 'N' or 'E', the Cholesky decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as

$A = U^H U$, if *uplo* = 'U', or
$A = L L^H$, if *uplo* = 'L',
where *U* is an upper triangular band matrix and *L* is a lower triangular band matrix.

3. If the leading *i*-by-*i* principal minor is not positive definite, then the routine returns with *info* = *i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, *info* = *n* + 1 is returned as a warning, but the routine still goes on to solve for *X* and compute error bounds as described below.

4. The system of equations is solved for *X* using the factored form of *A*.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix *X* is premultiplied by diag(*s*) so that it solves the original system before equilibration.

Input Parameters

fact — CHARACTER*1. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.

If *fact* = 'F': on entry, *afb* contains the factored form of *A*. If *equed* = 'Y', the matrix *A* has been equilibrated with scaling factors given by *s*.
ab and *afb* will not be modified.

If *fact* = 'N', the matrix *A* will be copied to *afb* and factored.
If *fact* = 'E', the matrix *A* will be equilibrated if necessary, then copied to *afb* and factored.

uplo — CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is stored and how A is factored:
If *uplo* = 'U', the array *ab* stores the upper triangular part of the matrix A, and A is factored as $U^H U$.
If *uplo* = 'L', the array *ab* stores the lower triangular part of the matrix A; A is factored as LL^H.

n — INTEGER. The order of matrix A ($n \ge 0$).

kd — INTEGER. The number of super-diagonals or sub-diagonals in the matrix A ($kd \ge 0$).

nrhs — INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \ge 0$).

ab, afb, b, work — REAL for spbsvx
DOUBLE PRECISION for dpbsvx
COMPLEX for cpbsvx
DOUBLE COMPLEX for zpbsvx.
Arrays: *ab*(*ldab*,*), *afb*(*ldab*,*), *b*(*ldb*,*), *work*(*).

The array *ab* contains the upper or lower triangle of the matrix A in band storage (see Matrix Storage Schemes).
If *fact* = 'F' and *equed* = 'Y', then *ab* must contain the equilibrated matrix diag(*s*)*A*diag(*s*). The second dimension of *ab* must be at least max(1, *n*).
The array *afb* is an input argument if *fact* = 'F' .
It contains the triangular factor U or L from the Cholesky factorization of the band matrix A in the same storage format as A. If *equed* = 'Y', then *afb* is the factored form of the equilibrated matrix A.
The second dimension of *afb* must be at least max(1,*n*).

The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

work(*) is a workspace array.
The dimension of *work* must be at least max(1,3**n*) for real flavors, and at least max(1,2**n*) for complex flavors.

ldab — INTEGER. The first dimension of *ab*; $ldab \ge kd+1$.

ldafb — INTEGER. The first dimension of *afb*; $ldafb \ge kd+1$.

ldb — INTEGER. The first dimension of *b*; $ldb \ge \max(1, n)$.

equed — `CHARACTER*1`. Must be `'N'` or `'Y'`.
equed is an input argument if *fact* = 'F' . It specifies the form of equilibration that was done:
If *equed* = `'N'`, no equilibration was done (always true if *fact* = 'N');
If *equed* = `'Y'`, equilibration was done and A has been replaced by diag(*s*)*A*diag(*s*).

s — `REAL` for single precision flavors;
`DOUBLE PRECISION` for double precision flavors.
Array, `DIMENSION` (*n*).
The array *s* contains the scale factors for A. This array is an input argument if *fact* = 'F' only; otherwise it is an output argument.
If *equed* = `'N'` , *s* is not accessed.
If *fact* = 'F' and *equed* = `'Y'`, each element of *s* must be positive.

ldx — `INTEGER`. The first dimension of the output array *x*; *ldx* ≥ max(1, *n*).

iwork — `INTEGER`.
Workspace array, `DIMENSION` at least max(1, *n*); used in real flavors only.

rwork — `REAL` for `cpbsvx`;
`DOUBLE PRECISION` for `zpbsvx`.
Workspace array, `DIMENSION` at least max(1, *n*); used in complex flavors only.

Output Parameters

x — `REAL` for `spbsvx`
`DOUBLE PRECISION` for `dpbsvx`
`COMPLEX` for `cpbsvx`
`DOUBLE COMPLEX` for `zpbsvx`.
Array, `DIMENSION` (*ldx*, *).

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix X to the *original* system of equations. Note that if *equed* = `'Y'`, A and B are modified on exit, and the solution to the equilibrated system is $\mathrm{diag}(s)^{-1} * X$.
The second dimension of *x* must be at least max(1,*nrhs*).

ab — On exit, if *fact* = 'E' and *equed* = 'Y', A is overwritten by diag(*s*)*A*diag(*s*)

afb — If *fact* = 'N' or `'E'`, then *afb* is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization $A=U^H U$ or $A=LL^H$ of the original matrix A(if *fact* = 'N'), or of the equilibrated matrix A (if *fact* = 'E'). See the description of *ab* for the form of the equilibrated matrix.

`b`	Overwritten by diag(`s`)**B* , if `equed` = `'Y'`; not changed if `equed` = `'N'`.
`s`	This array is an output argument if `fact` ≠ 'F' . See the description of `s` in *Input Arguments* section.
`rcond`	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If `rcond` is less than the machine precision (in particular, if `rcond` = 0), the matrix is singular to working precision. This condition is indicated by a return code of `info` > 0.
`ferr, berr`	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. Arrays, `DIMENSION` at least max(1,`nrhs`). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.
`equed`	If `fact` ≠ 'F' , then `equed` is an output argument. It specifies the form of equilibration that was done (see the description of `equed` in *Input Arguments* section).
`info`	`INTEGER`. If `info`=0, the execution is successful. If `info` = `-i`, the `i`th parameter had an illegal value. If `info` = `i`, and `i` ≤ `n`, the leading minor of order `i` (and hence the matrix *A* itself) is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed; `rcond` = 0 is returned. If `info` = `i`, and `i` = `n` +1, then *U* is nonsingular, but `rcond` is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of `rcond` would suggest.

?ptsv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive definite tridiagonal matrix A and multiple right-hand sides.

Syntax

```
call sptsv (n, nrhs, d, e, b, ldb, info)
```

```
call dptsv (n, nrhs, d, e, b, ldb, info)
call cptsv (n, nrhs, d, e, b, ldb, info)
call zptsv (n, nrhs, d, e, b, ldb, info)
```

Description

This routine solves for X the real or complex system of linear equations $AX = B$, where A is an *n*-by-*n* symmetric/Hermitian positive definite tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

A is factored as $A = L D L^H$, and the factored form of A is then used to solve the system of equations $AX = B$.

Input Parameters

n	`INTEGER`. The order of matrix A ($n \geq 0$).
nrhs	`INTEGER`. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
d	`REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. Array, dimension at least max(1, *n*). Contains the diagonal elements of the tridiagonal matrix A.
e, b	`REAL` for `sptsv` `DOUBLE PRECISION` for `dptsv` `COMPLEX` for `cptsv` `DOUBLE COMPLEX` for `zptsv`. Arrays: $e(n - 1)$, $b(ldb, *)$. The array *e* contains the $(n - 1)$ subdiagonal elements of A. The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).
ldb	`INTEGER`. The first dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

d	Overwritten by the *n* diagonal elements of the diagonal matrix D from the LDL^H factorization of A.

e — Overwritten by the $(n - 1)$ subdiagonal elements of the unit bidiagonal factor L from the factorization of A.

b — Overwritten by the solution matrix X.

info — `INTEGER`. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, the leading minor of order *i* (and hence the matrix A itself) is not positive definite, and the solution has not been computed. The factorization has not been completed unless *i* = *n*.

?ptsvx

Uses the factorization $A=LDL^H$ to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite tridiagonal matrix A, and provides error bounds on the solution.

Syntax

```
call sptsvx (fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr,
    berr, work, info)
call dptsvx (fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr,
    berr, work, info)
call cptsvx (fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr,
    berr, work, rwork, info)
call zptsvx (fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr,
    berr, work, rwork, info)
```

Description

This routine uses the Cholesky factorization $A=L\,D\,L^H$ to compute the solution to a real or complex system of linear equations $AX = B$, where A is a n-by-n symmetric or Hermitian positive definite tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?ptsvx` performs the following steps:

1. If *fact* = 'N', the matrix A is factored as $A = L D L^H$, where L is a unit lower bidiagonal matrix and D is diagonal. The factorization can also be regarded as having the form $A = U^H D U$.

2. If the leading *i*-by-*i* principal minor is not positive definite, then the routine returns with *info* = *i*. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, *info* = $n + 1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact	`CHARACTER*1`. Must be `'F'` or `'N'`. Specifies whether or not the factored form of the matrix A is supplied on entry. If *fact* = `'F'`: on entry, *df* and *ef* contain the factored form of A. Arrays *d*, *e*, *df*, and *ef* will not be modified. If *fact* = `'N'`, the matrix A will be copied to *df* and *ef* and factored.
n	`INTEGER`. The order of matrix A ($n \geq 0$).
nrhs	`INTEGER`. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
d, df, rwork	`REAL` for single precision flavors `DOUBLE PRECISION` for double precision flavors Arrays: *d*(*n*), *df*(*n*), *rwork*(*n*). The array *d* contains the *n* diagonal elements of the tridiagonal matrix A. The array *df* is an input argument if *fact* = `'F'` and on entry contains the *n* diagonal elements of the diagonal matrix D from the $L D L^H$ factorization of A. The array *rwork* is a workspace array used for complex flavors only.
e, ef, b, work	`REAL` for `sptsvx` `DOUBLE PRECISION` for `dptsvx` `COMPLEX` for `cptsvx` `DOUBLE COMPLEX` for `zptsvx`. Arrays: *e*(*n* - 1), *ef*(*n* - 1), *b*(*ldb*, *), *work*(*). The array *e* contains the ($n - 1$) subdiagonal elements of the tridiagonal matrix A.

The array *ef* is an input argument if *fact* = 'F' and on entry contains the (n - 1) subdiagonal elements of the unit bidiagonal factor L from the $L D L^H$ factorization of A.
The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations.
The array *work* is a workspace array. The dimension of *work* must be at least 2**n* for real flavors, and at least *n* for complex flavors.

ldb — INTEGER. The leading dimension of *b*; *ldb* ≥ max(1, *n*).

ldx — INTEGER. The leading dimension of *x*; *ldx* ≥ max(1, *n*).

Output Parameters

x — REAL for sptsvx
DOUBLE PRECISION for dptsvx
COMPLEX for cptsvx
DOUBLE COMPLEX for zptsvx.
Array, DIMENSION (*ldx*,*).

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix X to the system of equations. The second dimension of *x* must be at least max(1,*nrhs*).

df, ef — These arrays are output arguments if *fact* = 'N'.
See the description of *df, ef* in *Input Arguments* section.

rcond — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal condition number of the matrix A after equilibration (if done). If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr, berr — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(1,*nrhs*). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

info — INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, and *i* ≤ *n*, the leading minor of order *i* (and hence the matrix A itself) is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed; *rcond* = 0 is returned.
If *info* = *i*, and *i* = *n* +1, then U is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision.

Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

?sysv

Computes the solution to the system of linear equations with a real or complex symmetric matrix A and multiple right-hand sides.

Syntax

```
call ssysv (uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info)
call dsysv (uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info)
call csysv (uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info)
call zsysv (uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info)
```

Description

This routine solves for X the real or complex system of linear equations $AX = B$, where A is an n-by-n symmetric matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U D U^T$ or $A = L D L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $AX = B$.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates whether the upper or lower triangular part of A is stored and how A is factored:
If *uplo* = `'U'`, the array *a* stores the upper triangular part of the matrix A, and A is factored as UDU^T.
If *uplo* = `'L'`, the array *a* stores the lower triangular part of the matrix A; A is factored as LDL^T.

n — `INTEGER`. The order of matrix A ($n \geq 0$).

nrhs	`INTEGER`. The number of right-hand sides; the number of columns in *B* (*nrhs* ≥ 0).
a, b, work	`REAL` for `ssysv` `DOUBLE PRECISION` for `dsysv` `COMPLEX` for `csysv` `DOUBLE COMPLEX` for `zsysv`. Arrays: *a*(*lda*,*), *b*(*ldb*,*), *work*(*lwork*). The array *a* contains either the upper or the lower triangular part of the symmetric matrix *A* (see *uplo*). The second dimension of *a* must be at least max(1, *n*). The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*). *work*(*lwork*) is a workspace array.
lda	`INTEGER`. The first dimension of *a*; *lda* ≥ max(1, *n*).
ldb	`INTEGER`. The first dimension of *b*; *ldb* ≥ max(1, *n*).
lwork	`INTEGER`. The size of the *work* array (*lwork* ≥ 1) See *Application notes* for the suggested value of *lwork*.

Output Parameters

a	If *info* = 0, *a* is overwritten by the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L)* from the factorization of *A* as computed by `?sytrf`.
b	If *info* = 0, *b* is overwritten by the solution matrix *X*.
ipiv	`INTEGER`. Array, `DIMENSION` at least max(1,*n*). Contains details of the interchanges and the block structure of *D,* as determined by `?sytrf`. If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 diagonal block, and the *i*th row and column of *A* was interchanged with the *k*th row and column. If *uplo* = 'U' and *ipiv*(*i*) =*ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*-1, and (*i*-1)th row and column of *A* was interchanged with the *m*th row and column. If *uplo* = 'L' and *ipiv*(*i*) =*ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)th row and column of *A* was interchanged with the *m*th row and column.
work(1)	If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular, so the solution could not be computed.

Application Notes

For better performance, try using *lwork* = *n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use *lwork* =-1 for the first run. In this case, a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry *work*(1) of the *work* array , and no error message related to *lwork* is issued by XERBLA. On exit, examine *work*(1) and use this value for subsequent runs.

?sysvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a real or complex symmetric matrix A, and provides error bounds on the solution.

Syntax

```
call ssysvx (fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
    rcond, ferr, berr, work, lwork, iwork, info)
call dsysvx (fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
    rcond, ferr, berr, work, lwork, iwork, info)
call csysvx (fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
    rcond, ferr, berr, work, lwork, rwork, info)
call zsysvx (fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
    rcond, ferr, berr, work, lwork, rwork, info)
```

Description

This routine uses the diagonal pivoting factorization to compute the solution to a real or complex system of linear equations $AX = B$, where A is a *n*-by-*n* symmetric matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?sysvx` performs the following steps:

1. If *fact* = 'N', the diagonal pivoting method is used to factor the matrix A. The form of the factorization is $A = U D U^T$ or $A = L D L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with *info* = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, *info* = *n* + 1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact — `CHARACTER*1`. Must be `'F'` or `'N'`.

Specifies whether or not the factored form of the matrix A has been supplied on entry.

If *fact* = `'F'`: on entry, *af* and *ipiv* contain the factored form of A. Arrays *a*, *af*, and *ipiv* will not be modified.

If *fact* = `'N'`, the matrix A will be copied to *af* and factored.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates whether the upper or lower triangular part of A is stored and how A is factored:
If *uplo* = `'U'`, the array *a* stores the upper triangular part of the symmetric matrix A, and A is factored as UDU^T.
If *uplo* = `'L'`, the array *a* stores the lower triangular part of the symmetric matrix A; A is factored as LDL^T.

n — `INTEGER`. The order of matrix A ($n \ge 0$).

nrhs — `INTEGER`. The number of right-hand sides; the number of columns in B ($nrhs \ge 0$).

`a,af,b,work` REAL for ssysvx
DOUBLE PRECISION for dsysvx
COMPLEX for csysvx
DOUBLE COMPLEX for zsysvx.
Arrays: *a*(*lda*,*), *af*(*ldaf*,*), *b*(*ldb*,*), *work*(*).

The array *a* contains either the upper or the lower triangular part of the symmetric matrix A (see *uplo*).
The second dimension of *a* must be at least max(1,*n*).

The array *af* is an input argument if *fact* = 'F' . It contains he block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization $A = U D U^T$ or $A = L D L^T$ as computed by ?sytrf.
The second dimension of *af* must be at least max(1,*n*).

The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

work(*) is a workspace array of dimension (*lwork*).

lda INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).

ldaf INTEGER. The first dimension of *af*; *ldaf* ≥ max(1, *n*).

ldb INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

ipiv INTEGER.
Array, DIMENSION at least max(1,*n*).
The array *ipiv* is an input argument if *fact* = 'F' .
It contains details of the interchanges and the block structure of D, as determined by ?sytrf.
If *ipiv*(*i*) = $k > 0$, then d_{ii} is a 1-by-1 diagonal block, and the *i*th row and column of A was interchanged with the *k*th row and column.

If *uplo* = 'U' and *ipiv*(*i*) =*ipiv*(*i*-1) = -*m* < 0, then D has a 2-by-2 block in rows/columns *i* and *i*-1, and (*i*-1)th row and column of A was interchanged with the *m*th row and column.
If *uplo* = 'L' and *ipiv*(*i*) =*ipiv*(*i*+1) = -*m* < 0, then D has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)th row and column of A was interchanged with the *m*th row and column.

ldx INTEGER. The leading dimension of the output array *x*; *ldx* ≥ max(1, *n*).

lwork INTEGER. The size of the *work* array .
See *Application notes* for the suggested value of *lwork*.

iwork — `INTEGER`.
Workspace array, `DIMENSION` at least max(1, *n*); used in real flavors only.

rwork — `REAL` for `csysvx`;
`DOUBLE PRECISION` for `zsysvx`.
Workspace array, `DIMENSION` at least max(1, *n*); used in complex flavors only.

Output Parameters

x — `REAL` for `ssysvx`
`DOUBLE PRECISION` for `dsysvx`
`COMPLEX` for `csysvx`
`DOUBLE COMPLEX` for `zsysvx`.
Array, `DIMENSION` (*ldx*,*).

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the system of equations. The second dimension of *x* must be at least max(1,*nrhs*).

af, ipiv — These arrays are output arguments if *fact* = 'N'.
See the description of *af, ipiv* in *Input Arguments* section.

rcond — `REAL` for single precision flavors.
`DOUBLE PRECISION` for double precision flavors.
An estimate of the reciprocal condition number of the matrix *A*. If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr, berr — `REAL` for single precision flavors.
`DOUBLE PRECISION` for double precision flavors.
Arrays, `DIMENSION` at least max(1,*nrhs*). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

work(1) — If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — `INTEGER`. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, and $i \le n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.
If *info* = *i*, and $i = n + 1$, then *D* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision.

Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

Application Notes

For real flavors, *lwork* must be at least 3*n, and for complex flavors at least 2*n. For better performance, try using *lwork* = n**blocksize*, where *blocksize* is the optimal block size for `?sytrf`.

If you are in doubt how much workspace to supply, use *lwork* =-1 for the first run. In this case, a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry *work*(1) of the *work* array , and no error message related to *lwork* is issued by XERBLA. On exit, examine *work*(1) and use this value for subsequent runs.

?hesvx

Uses the diagonal pivoting factorization to compute the solution to the complex system of linear equations with a Hermitian matrix A, and provides error bounds on the solution.

Syntax

```
call chesvx (fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
    rcond, ferr, berr, work, lwork, rwork, info)
call zhesvx (fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
    rcond, ferr, berr, work, lwork, rwork, info)
```

Description

This routine uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations $AX = B$, where A is a n-by-n Hermitian matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?hesvx` performs the following steps:

1. If *fact* = 'N', the diagonal pivoting method is used to factor the matrix A. The form of the factorization is $A = U D U^H$ or $A = L D L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with *info* = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, *info* = *n* + 1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact — `CHARACTER*1`. Must be `'F'` or `'N'`.

Specifies whether or not the factored form of the matrix A has been supplied on entry.

If *fact* = `'F'`: on entry, *af* and *ipiv* contain the factored form of A. Arrays *a*, *af*, and *ipiv* will not be modified.

If *fact* = `'N'`, the matrix A will be copied to *af* and factored.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates whether the upper or lower triangular part of A is stored and how A is factored:
If *uplo* = `'U'`, the array *a* stores the upper triangular part of the Hermitian matrix A, and A is factored as UDU^H.
If *uplo* = `'L'`, the array *a* stores the lower triangular part of the Hermitian matrix A; A is factored as LDL^H.

n — `INTEGER`. The order of matrix A ($n \ge 0$).

nrhs — `INTEGER`. The number of right-hand sides; the number of columns in B ($nrhs \ge 0$).

a,af,b,work — `COMPLEX` for `chesvx`
`DOUBLE COMPLEX` for `zhesvx`.
Arrays: *a*(*lda*,*), *af*(*ldaf*,*), *b*(*ldb*,*), *work*(*).

The array *a* contains either the upper or the lower triangular part of the Hermitian matrix *A* (see *uplo*).
The second dimension of *a* must be at least max(1,*n*).

The array *af* is an input argument if *fact* = 'F' . It contains he block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization $A = U D U^H$ or $A = L D L^H$ as computed by ?hetrf.
The second dimension of *af* must be at least max(1,*n*).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).

work(*) is a workspace array of dimension (*lwork*).

lda INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).

ldaf INTEGER. The first dimension of *af*; *ldaf* ≥ max(1, *n*).

ldb INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

ipiv INTEGER.
Array, DIMENSION at least max(1,*n*).
The array *ipiv* is an input argument if *fact* = 'F' .
It contains details of the interchanges and the block structure of *D*, as determined by ?hetrf.
If *ipiv(i)* = *k* > 0, then d_{ii} is a 1-by-1 diagonal block, and the *i*th row and column of *A* was interchanged with the *k*th row and column.

If *uplo* = 'U' and *ipiv(i)* =*ipiv(i-1)* = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i-1*, and (*i-1*)th row and column of *A* was interchanged with the *m*th row and column.
If *uplo* = 'L' and *ipiv(i)* =*ipiv(i+1)* = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i+1*, and (*i+1*)th row and column of *A* was interchanged with the *m*th row and column.

ldx INTEGER. The leading dimension of the output array *x*; *ldx* ≥ max(1, *n*).

lwork INTEGER. The size of the *work* array .
See *Application notes* for the suggested value of *lwork*.

rwork REAL for chesvx;
DOUBLE PRECISION for zhesvx.
Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

x — COMPLEX for chesvx
DOUBLE COMPLEX for zhesvx.
Array, DIMENSION (*ldx*,*).

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the system of equations. The second dimension of *x* must be at least max(1,*nrhs*).

af, ipiv — These arrays are output arguments if *fact* = 'N'.
See the description of *af, ipiv* in *Input Arguments* section.

rcond — REAL for chesvx;
DOUBLE PRECISION for zhesvx.
An estimate of the reciprocal condition number of the matrix *A*. If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr, berr — REAL for chesvx;
DOUBLE PRECISION for zhesvx.
Arrays, DIMENSION at least max(1,*nrhs*). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

work(1) — If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, and $i \le n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.
If *info* = *i*, and $i = n + 1$, then *D* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

Application Notes

The value of *lwork* must be at least 2*n. For better performance, try using *lwork* = *n***blocksize*, where *blocksize* is the optimal block size for ?hetrf.

If you are in doubt how much workspace to supply, use `lwork` =-1 for the first run. In this case, a workspace query is assumed; the routine only calculates the optimal size of the `work` array, returns this value as the first entry `work(1)` of the `work` array , and no error message related to `lwork` is issued by XERBLA. On exit, examine `work(1)` and use this value for subsequent runs.

?hesv

Computes the solution to the system of linear equations with a Hermitian matrix A and multiple right-hand sides.

Syntax

```
call chesv (uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info)
call zhesv (uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info)
```

Description

This routine solves for X the real or complex system of linear equations $AX = B$, where A is an *n*-by-*n* symmetric matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U D U^H$ or $A = L D L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $AX = B$.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If *uplo* = `'U'`, the array *a* stores the upper triangular part of the matrix A, and A is factored as UDU^H. If *uplo* = `'L'`, the array *a* stores the lower triangular part of the matrix A; A is factored as LDL^H.
n	`INTEGER`. The order of matrix A ($n \ge 0$).
nrhs	`INTEGER`. The number of right-hand sides; the number of columns in B ($nrhs \ge 0$).

a, b, work	COMPLEX for chesv DOUBLE COMPLEX for zhesv. Arrays: *a*(*lda*,*), *b*(*ldb*,*), *work*(*lwork*). The array *a* contains either the upper or the lower triangular part of the Hermitian matrix *A* (see *uplo*). The second dimension of *a* must be at least max(1, *n*). The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*). *work*(*lwork*) is a workspace array.
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).
lwork	INTEGER. The size of the *work* array (*lwork* ≥ 1) See *Application notes* for the suggested value of *lwork*.

Output Parameters

a	If *info* = 0, *a* is overwritten by the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L)* from the factorization of *A* as computed by ?hetrf.
b	If *info* = 0, *b* is overwritten by the solution matrix *X*.
ipiv	INTEGER. Array, DIMENSION at least max(1,*n*). Contains details of the interchanges and the block structure of *D,* as determined by ?hetrf. If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 diagonal block, and the *i*th row and column of *A* was interchanged with the *k*th row and column. If *uplo* = 'U' and *ipiv*(*i*) =*ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*-1, and (*i*-1)th row and column of *A* was interchanged with the *m*th row and column. If *uplo* = 'L' and *ipiv*(*i*) =*ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)th row and column of *A* was interchanged with the *m*th row and column.
work(1)	If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular, so the solution could not be computed.

Application Notes

For better performance, try using `lwork` = `n*`*blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use `lwork` =-1 for the first run. In this case, a workspace query is assumed; the routine only calculates the optimal size of the `work` array, returns this value as the first entry `work(1)` of the `work` array , and no error message related to `lwork` is issued by XERBLA. On exit, examine `work(1)` and use this value for subsequent runs.

?spsv

Computes the solution to the system of linear equations with a real or complex symmetric matrix A stored in packed format, and multiple right-hand sides.

Syntax

```
call sspsv (uplo, n, nrhs, ap, ipiv, b, ldb, info)
call dspsv (uplo, n, nrhs, ap, ipiv, b, ldb, info)
call cspsv (uplo, n, nrhs, ap, ipiv, b, ldb, info)
call zspsv (uplo, n, nrhs, ap, ipiv, b, ldb, info)
```

Description

This routine solves for X the real or complex system of linear equations $AX = B$, where A is an n-by-n symmetric matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U D U^T$ or $A = L D L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $AX = B$.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If *uplo* = 'U', the array *ap* stores the upper triangular part of the matrix A, and A is factored as UDU^T.
If *uplo* = 'L', the array *ap* stores the lower triangular part of the matrix A; A is factored as LDL^T.

n — INTEGER. The order of matrix A ($n \geq 0$).

nrhs — INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).

ap, b — REAL for sspsv
DOUBLE PRECISION for dspsv
COMPLEX for cspsv
DOUBLE COMPLEX for zspsv.
Arrays: *ap*(*), *b*(*ldb*,*)
The dimension of *ap* must be at least max(1,n(n+1)/2).
The array *ap* contains the factor U or L, as specified by *uplo*, in *packed storage* (see Matrix Storage Schemes).
The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations.
The second dimension of *b* must be at least max(1,*nrhs*).

ldb — INTEGER. The first dimension of *b*; $ldb \geq$ max(1, n).

Output Parameters

ap — The block-diagonal matrix D and the multipliers used to obtain the factor U (or L) from the factorization of A as computed by ?sptrf, stored as a packed triangular matrix in the same storage format as A.

b — If *info* = 0, *b* is overwritten by the solution matrix X.

ipiv — INTEGER.
Array, DIMENSION at least max(1,n).
Contains details of the interchanges and the block structure of D, as determined by ?sptrf.
If *ipiv*(*i*) = $k > 0$, then d_{ii} is a 1-by-1 block, and the *i*th row and column of A was interchanged with the *k*th row and column.

If *uplo* = 'U' and *ipiv*(*i*) =*ipiv*(*i*-1) = -$m < 0$, then D has a 2-by-2 block in rows/columns *i* and *i*-1, and (*i*-1)th row and column of A was interchanged with the *m*th row and column.

If *uplo* = 'L' and *ipiv*(*i*) =*ipiv*(*i*+1) = -$m < 0$, then D has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)th row and column of A was interchanged with the *m*th row and column.

info INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular, so the solution could not be computed.

?spsvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a real or complex symmetric matrix A stored in packed format, and provides error bounds on the solution.

Syntax

```
call sspsvx (fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond,
    ferr, berr, work, iwork, info)
call dspsvx (fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond,
    ferr, berr, work, iwork, info)
call cspsvx (fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond,
    ferr, berr, work, rwork, info)
call zspsvx (fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond,
    ferr, berr, work, rwork, info)
```

Description

This routine uses the diagonal pivoting factorization to compute the solution to a real or complex system of linear equations $AX = B$, where A is a n-by-n symmetric matrix stored in packed format, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?spsvx performs the following steps:

1. If *fact* = 'N', the diagonal pivoting method is used to factor the matrix A. The form of the factorization is $A = U D U^T$ or $A = L D L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and *D* is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with *info* = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, *info* = $n + 1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact — CHARACTER*1. Must be 'F' or 'N'.

Specifies whether or not the factored form of the matrix A has been supplied on entry.

If *fact* = 'F': on entry, *afp* and *ipiv* contain the factored form of A. Arrays *ap*, *afp*, and *ipiv* will not be modified.

If *fact* = 'N', the matrix A will be copied to *afp* and factored.

uplo — CHARACTER*1. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of A is stored and how A is factored:
If *uplo* = 'U', the array *ap* stores the upper triangular part of the symmetric matrix A, and A is factored
as UDU^T.
If *uplo* = 'L', the array *ap* stores the lower triangular part of the symmetric matrix A; A is factored as LDL^T.

n — INTEGER. The order of matrix A ($n \geq 0$).

nrhs — INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).

ap,afp,b,work — REAL for sspsvx
DOUBLE PRECISION for dspsvx
COMPLEX for cspsvx
DOUBLE COMPLEX for zspsvx.
Arrays: *ap*(*), *afp*(*), *b*(*ldb*,*), *work* (*).

The array *ap* contains the upper or lower triangle of the symmetric matrix A in *packed storage* (see Matrix Storage Schemes).

The array *afp* is an input argument if *fact* = 'F'. It contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization
$A = U D U^T$ or $A = L D L^T$ as computed by ?sptrf, in the same storage format as *A*.
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.
work (*) is a workspace array.

The dimension of arrays *ap* and *afp* must be at least max(1,*n*(*n*+1)/2); the second dimension of *b* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.

ldb — INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

ipiv — INTEGER.
Array, DIMENSION at least max(1,*n*).
The array *ipiv* is an input argument if *fact* = 'F'.
It contains details of the interchanges and the block structure of *D,* as determined by ?sptrf.
If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 diagonal block, and the *i*th row and column of *A* was interchanged with the *k*th row and column.

If *uplo* = 'U' and *ipiv*(*i*) =*ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*-1, and (*i*-1)th row and column of *A* was interchanged with the *m*th row and column.
If *uplo* = 'L' and *ipiv*(*i*) =*ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)th row and column of *A* was interchanged with the *m*th row and column.

ldx — INTEGER. The leading dimension of the output array *x*; *ldx* ≥ max(1, *n*).

iwork — INTEGER.
Workspace array, DIMENSION at least max(1, *n*); used in real flavors only.

rwork — REAL for cspsvx;
DOUBLE PRECISION for zspsvx.
Workspace array, DIMENSION at least max(1, *n*); used in complex flavors only.

Output Parameters

x — `REAL` for `sspsvx`
`DOUBLE PRECISION` for `dspsvx`
`COMPLEX` for `cspsvx`
`DOUBLE COMPLEX` for `zspsvx`.
Array, `DIMENSION (`*ldx*`,*)`.

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the system of equations. The second dimension of *x* must be at least max(1,*nrhs*).

afp, ipiv — These arrays are output arguments if *fact* = 'N'.
See the description of *afp, ipiv* in *Input Arguments* section.

rcond — `REAL` for single precision flavors.
`DOUBLE PRECISION` for double precision flavors.
An estimate of the reciprocal condition number of the matrix *A*. If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr, berr — `REAL` for single precision flavors.
`DOUBLE PRECISION` for double precision flavors.
Arrays, `DIMENSION` at least max(1,*nrhs*). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

info — `INTEGER`. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, and $i \le n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.
If *info* = *i*, and $i = n + 1$, then *D* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

?hpsvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a Hermitian matrix A stored in packed format, and provides error bounds on the solution.

Syntax

```
call chpsvx (fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond,
    ferr, berr, work, rwork, info)
call zhpsvx (fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond,
    ferr, berr, work, rwork, info)
```

Description

This routine uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations $AX = B$, where A is a *n*-by-*n* Hermitian matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?hpsvx` performs the following steps:

1. If *fact* = 'N', the diagonal pivoting method is used to factor the matrix A. The form of the factorization is $A = U D U^H$ or $A = L D L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with *info* = i. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, *info* = *n* + 1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

3. The system of equations is solved for X using the factored form of A.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact — `CHARACTER*1`. Must be `'F'` or `'N'`.

Specifies whether or not the factored form of the matrix *A* has been supplied on entry.

If *fact* = `'F'`: on entry, *afp* and *ipiv* contain the factored form of *A*. Arrays *ap*, *afp*, and *ipiv* will not be modified.

If *fact* = `'N'`, the matrix *A* will be copied to *afp* and factored.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates whether the upper or lower triangular part of *A* is stored and how *A* is factored:
If *uplo* = `'U'`, the array *ap* stores the upper triangular part of the Hermitian matrix *A*, and *A* is factored as UDU^H.
If *uplo* = `'L'`, the array *ap* stores the lower triangular part of the Hermitian matrix *A*; *A* is factored as LDL^H.

n — `INTEGER`. The order of matrix *A* ($n \geq 0$).

nrhs — `INTEGER`. The number of right-hand sides; the number of columns in *B* ($nrhs \geq 0$).

ap,afp,b,work — `COMPLEX` for `chpsvx`
`DOUBLE COMPLEX` for `zhpsvx`.
Arrays: *ap*(*), *afp*(*), *b*(*ldb*,*), *work* (*).

The array *ap* contains the upper or lower triangle of the Hermitian matrix A in *packed storage* (see Matrix Storage Schemes).

The array *afp* is an input argument if *fact* = 'F'. It contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization
$A = U D U^H$ or $A = L D L^H$ as computed by `?hptrf`, in the same storage format as *A*.
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.
work (*) is a workspace array.

The dimension of arrays *ap* and *afp* must be at least max(1,*n*(*n*+1)/2); the second dimension of *b* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1, 2**n*) .

ldb — `INTEGER`. The first dimension of *b*; $ldb \geq \max(1, n)$.

ipiv — `INTEGER`.
Array, `DIMENSION` at least max(1,*n*).
The array *ipiv* is an input argument if *fact* = 'F'.
It contains details of the interchanges and the block structure of *D*, as determined by ?hptrf.
If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 diagonal block, and the *i*th row and column of *A* was interchanged with the *k*th row and column.

If *uplo* = 'U' and *ipiv*(*i*) =*ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*-1, and (*i*-1)th row and column of *A* was interchanged with the *m*th row and column.
If *uplo* = 'L' and *ipiv*(*i*) =*ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)th row and column of *A* was interchanged with the *m*th row and column.

ldx — `INTEGER`. The leading dimension of the output array *x*; *ldx* ≥ max(1, *n*).

rwork — `REAL` for `chpsvx`;
`DOUBLE PRECISION` for `zhpsvx`.
Workspace array, `DIMENSION` at least max(1, *n*).

Output Parameters

x — `COMPLEX` for `chpsvx`
`DOUBLE COMPLEX` for `zhpsvx`.
Array, `DIMENSION` (*ldx*, *).

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the system of equations. The second dimension of *x* must be at least max(1,*nrhs*).

afp, ipiv — These arrays are output arguments if *fact* = 'N'.
See the description of *afp, ipiv* in *Input Arguments* section.

rcond — `REAL` for `chpsvx`;
`DOUBLE PRECISION` for `zhpsvx`.
An estimate of the reciprocal condition number of the matrix *A*. If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr, berr — `REAL` for `chpsvx`;
`DOUBLE PRECISION` for `zhpsvx`.
Arrays, `DIMENSION` at least max(1,*nrhs*). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

info — `INTEGER`. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, and $i \le n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.
If *info* = *i*, and $i = n + 1$, then *D* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

?hpsv

Computes the solution to the system of linear equations with a Hermitian matrix A stored in packed format, and multiple right-hand sides.

Syntax

```
call chpsv (uplo, n, nrhs, ap, ipiv, b, ldb, info)
call zhpsv (uplo, n, nrhs, ap, ipiv, b, ldb, info)
```

Description

This routine solves for *X* the system of linear equations $AX = B$, where A is an *n*-by-*n* Hermitian matrix stored in packed format, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The diagonal pivoting method is used to factor *A* as $A = U D U^H$ or $A = L D L^H$, where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of *A* is then used to solve the system of equations $AX = B$.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates whether the upper or lower triangular part of *A* is stored and how *A* is factored:

	If *uplo* = 'U', the array *ap* stores the upper triangular part of the matrix A, and A is factored as UDU^H. If *uplo* = 'L', the array *ap* stores the lower triangular part of the matrix A; A is factored as LDL^H.
n	INTEGER. The order of matrix A ($n \geq 0$).
nrhs	INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
ap, b	COMPLEX for chpsv DOUBLE COMPLEX for zhpsv. Arrays: *ap*(*), *b*(*ldb*,*) The dimension of *ap* must be at least max(1,n(n+1)/2). The array *ap* contains the factor U or L, as specified by *uplo*, in *packed storage* (see Matrix Storage Schemes). The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).
ldb	INTEGER. The first dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

ap	The block-diagonal matrix D and the multipliers used to obtain the factor U (or L) from the factorization of A as computed by ?hptrf, stored as a packed triangular matrix in the same storage format as A.
b	If *info* = 0, *b* is overwritten by the solution matrix X.
ipiv	INTEGER. Array, DIMENSION at least max(1,n). Contains details of the interchanges and the block structure of D, as determined by ?hptrf. If *ipiv*(*i*) = $k > 0$, then d_{ii} is a 1-by-1 block, and the *i*th row and column of A was interchanged with the *k*th row and column. If *uplo* = 'U' and *ipiv*(*i*) =*ipiv*(*i*-1) = $-m < 0$, then D has a 2-by-2 block in rows/columns *i* and *i*-1, and (*i*-1)th row and column of A was interchanged with the *m*th row and column. If *uplo* = 'L' and *ipiv*(*i*) =*ipiv*(*i*+1) = $-m < 0$, then D has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)th row and column of A was interchanged with the *m*th row and column.

info — `INTEGER`. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular, so the solution could not be computed.

LAPACK Routines: Least Squares and Eigenvalue Problems

4

This chapter describes the Intel® Math Kernel Library implementation of routines from the LAPACK package that are used for solving linear least-squares problems, eigenvalue and singular value problems, as well as performing a number of related computational tasks.

Sections in this chapter include descriptions of LAPACK computational routines and driver routines.

For full reference on LAPACK routines and related information see [*LUG*].

Least-Squares Problems. A typical *least-squares problem* is as follows: given a matrix A and a vector b, find the vector x that minimizes the sum of squares $\Sigma_i ((Ax)_i - b_i)^2$ or, equivalently, find the vector x that minimizes the 2-norm $\|Ax - b\|_2$.

In the most usual case, A is an m by n matrix with $m \geq n$ and $\text{rank}(A) = n$. This problem is also referred to as finding the *least-squares solution* to an *overdetermined* system of linear equations (here we have more equations than unknowns). To solve this problem, you can use the *QR* factorization of the matrix A (see *QR Factorization* on page 4-5).

If $m < n$ and $\text{rank}(A) = m$, there exist an infinite number of solutions x which exactly satisfy $Ax = b$, and thus minimize the norm $\|Ax - b\|_2$. In this case it is often useful to find the unique solution that minimizes $\|x\|_2$. This problem is referred to as finding the *minimum-norm solution* to an *underdetermined* system of linear equations (here we have more unknowns than equations). To solve this problem, you can use the *LQ* factorization of the matrix A (see *LQ Factorization* on page 4-6).

In the general case you may have a *rank-deficient least-squares problem,* with $\text{rank}(A) < \min(m, n)$: find the *minimum-norm least-squares solution* that minimizes both $\|x\|_2$ and $\|Ax - b\|_2$. In this case (or when the rank of A is in doubt) you can use the *QR* factorization with pivoting or *singular value decomposition* (see page 4-68).

Eigenvalue Problems (from German *eigen* "own") are stated as follows: given a matrix A, find the *eigenvalues* λ and the corresponding *eigenvectors* z that satisfy the equation

$$Az = \lambda z \quad \text{(right eigenvectors } z\text{)}$$

or the equation

$$z^H A = \lambda z^H \quad \text{(left eigenvectors } z\text{).}$$

If A is a real symmetric or complex Hermitian matrix, the above two equations are equivalent, and the problem is called a *symmetric* eigenvalue problem. Routines for solving this type of problems are described in the section *Symmetric Eigenvalue Problems* (see page 4-95).

Routines for solving eigenvalue problems with nonsymmetric or non-Hermitian matrices are described in the section *Nonsymmetric Eigenvalue Problems* (see page 4-162).

The library also includes routines that handle *generalized symmetric- definite eigenvalue problems*: find the eigenvalues λ and the corresponding eigenvectors x that satisfy one of the following equations:

$$Az = \lambda Bz, \quad ABz = \lambda z, \quad \text{or} \quad BAz = \lambda z$$

where A is symmetric or Hermitian, and B is symmetric positive-definite or Hermitian positive-definite. Routines for reducing these problems to standard symmetric eigenvalue problems are described in the section *Generalized Symmetric-Definite Eigenvalue Problems* (see page 4-147).

* * *

To solve a particular problem, you usually call several computational routines. Sometimes you need to combine the routines of this chapter with other LAPACK routines described in Chapter 3 as well as with BLAS routines (Chapter 2).

For example, to solve a set of least-squares problems minimizing $\|Ax - b\|_2$ for all columns b of a given matrix B (where A and B are real matrices), you can call `?geqrf` to form the factorization $A = QR$, then call `?ormqr` to compute $C = Q^H B$, and finally call the BLAS routine `?trsm` to solve for X the system of equations $RX = C$.

Another way is to call an appropriate driver routine that performs several tasks in one call. For example, to solve the least-squares problem the driver routine `?gels` can be used.

WARNING. LAPACK routines expect that input matrices do not contain `INF` or `NaN` values. When input data is inappropriate for LAPACK, problems may arise, including possible hangs.

Routine Naming Conventions

For each routine in this chapter, you can use the LAPACK name.

LAPACK names have the structure `xyyzzz`, which is described below.

The initial letter `x` indicates the data type:

`s` real, single precision`c` complex, single precision
`d` real, double precision`z` complex, double precision

The second and third letters `yy` indicate the matrix type and storage scheme:

`bd` bidiagonal matrix
`ge` general matrix
`gb` general band matrix
`hs` upper Hessenberg matrix
`or` (real) orthogonal matrix
`op` (real) orthogonal matrix (packed storage)
`un` (complex) unitary matrix
`up` (complex) unitary matrix (packed storage)
`pt` symmetric or Hermitian positive-definite tridiagonal matrix
`sy` symmetric matrix
`sp` symmetric matrix (packed storage)
`sb` (real) symmetric band matrix
`st` (real) symmetric tridiagonal matrix
`he` Hermitian matrix
`hp` Hermitian matrix (packed storage)
`hb` (complex) Hermitian band matrix
`tr` triangular or quasi-triangular matrix.

The last three letters `zzz` indicate the computation performed, for example:

`qrf` form the *QR* factorization
`lqf` form the *LQ* factorization.

Thus, the routine `sgeqrf` forms the *QR* factorization of general real matrices in single precision; the corresponding routine for complex matrices is `cgeqrf`.

Matrix Storage Schemes

LAPACK routines use the following matrix storage schemes:

- *Full storage*: a matrix A is stored in a two-dimensional array `a`, with the matrix element a_{ij} stored in the array element `a(i,j)`.
- *Packed storage* scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly: the upper or lower triangle of the matrix is packed by columns in a one-dimensional array.
- *Band storage*: an `m` by `n` band matrix with `kl` sub-diagonals and `ku` super-diagonals is stored compactly in a two-dimensional array `ab` with `kl`+`ku`+1 rows and `n` columns. Columns of the matrix are stored in the corresponding columns of the array, and *diagonals* of the matrix are stored in rows of the array.

In Chapters 3 and 4, arrays that hold matrices in packed storage have names ending in `p`; arrays with matrices in band storage have names ending in `b`.

For more information on matrix storage schemes, see "Matrix Arguments" in Appendix B.

Mathematical Notation

In addition to the mathematical notation used in previous chapters, descriptions of routines in this chapter use the following notation:

λ_i — *Eigenvalues* of the matrix A (for the definition of eigenvalues, see *Eigenvalue Problems* on page 4-2).

σ_i — *Singular values* of the matrix A. They are equal to square roots of the eigenvalues of $A^H A$. (For more information, see Singular Value Decomposition).

$||x||_2$ — The *2-norm* of the vector x: $||x||_2 = (\Sigma_i |x_i|^2)^{1/2} = ||x||_E$.

$||A||_2$ — The *2-norm* (or *spectral norm*) of the matrix A.
$||A||_2 = \max_i \sigma_i$, $||A||_2^2 = \max_{|x|=1}(Ax \cdot Ax)$.

$||A||_E$ — The *Euclidean norm* of the matrix A: $||A||_E^2 = \Sigma_i \Sigma_j |a_{ij}|^2$ (for vectors, the Euclidean norm and the 2-norm are equal: $||x||_E = ||x||_2$).

q(x, y) — The *acute angle between vectors x* and y:
cos q(x, y) = $|x \cdot y|$ / ($||x||_2$ $||y||_2$).

Computational Routines

In the sections that follow, the descriptions of LAPACK computational routines are given. These routines perform distinct computational tasks that can be used for:

Orthogonal Factorizations
Singular Value Decomposition
Symmetric Eigenvalue Problems
Generalized Symmetric-Definite Eigenvalue Problems
Nonsymmetric Eigenvalue Problems
Generalized Nonsymmetric Eigenvalue Problems
Generalized Singular Value Decomposition

See also the respective driver routines.

Orthogonal Factorizations

This section describes the LAPACK routines for the *QR (RQ)* and *LQ (QL)* factorization of matrices. Routines for the *RZ* factorization as well as for generalized *QR* and *RQ* factorizations are also included.

QR Factorization. Assume that A is an m by n matrix to be factored.
If $m \geq n$, the *QR* factorization is given by

$$A = Q\begin{pmatrix} R \\ 0 \end{pmatrix} = (Q_1, Q_2)\begin{pmatrix} R \\ 0 \end{pmatrix}$$

where R is an n by n upper triangular matrix with real diagonal elements, and Q is an m by m orthogonal (or unitary) matrix.

You can use the *QR* factorization for solving the following least-squares problem: minimize $\|Ax - b\|_2$ where A is a full-rank m by n matrix ($m \geq n$). After factoring the matrix, compute the solution x by solving $Rx = (Q_1)^T b$.

If $m < n$, the *QR* factorization is given by

$$A = QR = Q(R_1 R_2)$$

where R is trapezoidal, R_1 is upper triangular and R_2 is rectangular.

The LAPACK routines do not form the matrix Q explicitly. Instead, Q is represented as a product of min(m, n) *elementary reflectors*. Routines are provided to work with Q in this representation.

LQ Factorization of an m by n matrix A is as follows. If $m \le n$,

$$A = (L, 0)Q = (L, 0)\begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix} = LQ_1$$

where L is an m by m lower triangular matrix with real diagonal elements, and Q is an n by n orthogonal (or unitary) matrix.

If $m > n$, the LQ factorization is

$$A = \begin{pmatrix} L_1 \\ L_2 \end{pmatrix} Q$$

where L_1 is an n by n lower triangular matrix, L_2 is rectangular, and Q is an n by n orthogonal (or unitary) matrix.

You can use the LQ factorization to find the minimum-norm solution of an underdetermined system of linear equations $Ax = b$ where A is an m by n matrix of rank m ($m < n$). After factoring the matrix, compute the solution vector x as follows: solve $Ly = b$ for y, and then compute $x = (Q_1)^H y$.

Table 5-1 lists LAPACK routines that perform orthogonal factorization of matrices.

Table 4-1 Computational Routines for Orthogonal Factorization

Matrix type, factorization	Factorize without pivoting	Factorize with pivoting	Generate matrix Q	Apply matrix Q
general matrices, QR factorization	?geqrf	?geqpf ?geqp3	?orgqr ?ungqr	?ormqr ?unmqr
general matrices, RQ factorization	?gerqf		?orgrq ?ungrq	?ormrq ?unmrq
general matrices, LQ factorization	?gelqf		?orglq ?unglq	?ormlq ?unmlq
general matrices, QL factorization	?geqlf		?orgql ?ungql	?ormql ?unmql
trapezoidal matrices, RZ factorization	?tzrzf			?ormrz ?unmrz
pair of matrices, generalized QR factorization	?ggqrf			
pair of matrices, generalized RQ factorization	?ggrqf			

?geqrf

Computes the QR factorization of a general m by n matrix.

Syntax

```
call sgeqrf ( m, n, a, lda, tau, work, lwork, info )
call dgeqrf ( m, n, a, lda, tau, work, lwork, info )
call cgeqrf ( m, n, a, lda, tau, work, lwork, info )
call zgeqrf ( m, n, a, lda, tau, work, lwork, info )
```

Description

The routine forms the *QR* factorization of a general `m` by `n` matrix *A* (see *Orthogonal Factorizations* on page 4-5). No pivoting is performed.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(`m`, `n`) *elementary reflectors*. Routines are provided to work with *Q* in this representation.

Input Parameters

m	`INTEGER`. The number of rows in the matrix *A* ($m \geq 0$).
n	`INTEGER`. The number of columns in *A* ($n \geq 0$).
a, work	`REAL` for `sgeqrf` `DOUBLE PRECISION` for `dgeqrf` `COMPLEX` for `cgeqrf` `DOUBLE COMPLEX` for `zgeqrf`. Arrays: `a(lda,*)` contains the matrix *A*. The second dimension of `a` must be at least max(1, `n`). `work(lwork)` is a workspace array.
lda	`INTEGER`. The first dimension of `a`; at least max(1, `m`).
lwork	`INTEGER`. The size of the `work` array ($lwork \geq n$) See Application notes for the suggested value of `lwork`.

Output Parameters

a — Overwritten by the factorization data as follows:

If $m \ge n$, the elements below the diagonal are overwritten by the details of the unitary matrix *Q*, and the upper triangle is overwritten by the corresponding elements of the upper triangular matrix *R*.

If $m < n$, the strictly lower triangular part is overwritten by the details of the unitary matrix *Q*, and the remaining elements are overwritten by the corresponding elements of the *m* by *n* upper trapezoidal matrix *R*.

tau — `REAL` for `sgeqrf`
`DOUBLE PRECISION` for `dgeqrf`
`COMPLEX` for `cgeqrf`
`DOUBLE COMPLEX` for `zgeqrf`.
Array, `DIMENSION` at least max (1, min(*m*, *n*)).
Contains additional information on the matrix *Q*.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* =*n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The computed factorization is the exact factorization of a matrix $A + E$, where $\|E\|_2 = O(\varepsilon)\,\|A\|_2$.

The approximate number of floating-point operations for real flavors is

$(4/3)n^3$ if $m = n$,

$(2/3)n^2(3m-n)$ if $m > n$,

$(2/3)m^2(3n-m)$ if $m < n$.

The number of operations for complex flavors is 4 times greater.

To solve a set of least-squares problems minimizing $||Ax - b||_2$ for all columns b of a given matrix B, you can call the following:

`?geqrf` (this routine) to factorize $A = QR$;

`?ormqr` to compute $C = Q^T B$ (for real matrices);

`?unmqr` to compute $C = Q^H B$ (for complex matrices);

`?trsm` (a BLAS routine) to solve $RX = C$.

(The columns of the computed X are the least-squares solution vectors x.)

To compute the elements of Q explicitly, call

`?orgqr` (for real matrices)

`?ungqr` (for complex matrices).

?geqpf

Computes the QR factorization of a general m by n matrix with pivoting.

Syntax

```
call sgeqpf ( m, n, a, lda, jpvt, tau, work, info )
call dgeqpf ( m, n, a, lda, jpvt, tau, work, info )
call cgeqpf ( m, n, a, lda, jpvt, tau, work, rwork, info )
call zgeqpf ( m, n, a, lda, jpvt, tau, work, rwork, info )
```

Description

This routine is deprecated and has been replaced by routine `?geqp3`.

The routine `?geqpf` forms the QR factorization of a general m by n matrix A with column pivoting: $AP = QR$ (see *Orthogonal Factorizations* on page 4-5). Here P denotes an n by n permutation matrix.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of min(m, n) *elementary reflectors*. Routines are provided to work with Q in this representation.

Input Parameters

m	`INTEGER`. The number of rows in the matrix A ($m \ge 0$).
n	`INTEGER`. The number of columns in A ($n \ge 0$).
a, work	`REAL` for `sgeqpf` `DOUBLE PRECISION` for `dgeqpf` `COMPLEX` for `cgeqpf` `DOUBLE COMPLEX` for `zgeqpf`. Arrays: *a*(*lda*,*) contains the matrix A. The second dimension of *a* must be at least max(1, *n*). *work*(*lwork*) is a workspace array.
lda	`INTEGER`. The first dimension of *a*; at least max(1, *m*).
lwork	`INTEGER`. The size of the *work* array; must be at least max(1, 3*n*).
jpvt	`INTEGER`. Array, `DIMENSION` at least max(1, *n*). On entry, if *jpvt*(*i*)> 0, the *i*th column of A is moved to the beginning of AP before the computation, and fixed in place during the computation. If *jpvt*(*i*) = 0, the *i*th column of A is a free column (that is, it may be interchanged during the computation with any other free column).
rwork	`REAL` for `cgeqpf` `DOUBLE PRECISION` for `zgeqpf`. A workspace array, `DIMENSION` at least max(1, 2**n*).

Output Parameters

a	Overwritten by the factorization data as follows: If $m \ge n$, the elements below the diagonal are overwritten by the details of the unitary (orthogonal) matrix Q, and the upper triangle is overwritten by the corresponding elements of the upper triangular matrix R. If $m < n$, the strictly lower triangular part is overwritten by the details of the matrix Q, and the remaining elements are overwritten by the corresponding elements of the m by n upper trapezoidal matrix R.
tau	`REAL` for `sgeqpf` `DOUBLE PRECISION` for `dgeqpf` `COMPLEX` for `cgeqpf`

	`DOUBLE COMPLEX` for `zgeqpf`. Array, `DIMENSION` at least max (1, min(*m*, *n*)). Contains additional information on the matrix *Q*.
`jpvt`	Overwritten by details of the permutation matrix *P* in the factorization $AP = QR$. More precisely, the columns of *AP* are the columns of *A* in the following order: *`jpvt`*`(1)`, *`jpvt`*`(2)`, `...`, *`jpvt`*`(`*`n`*`)`.
`info`	`INTEGER`. If *`info`* = 0, the execution is successful. If *`info`* = `-`*`i`*, the *`i`*th parameter had an illegal value.

Application Notes

The computed factorization is the exact factorization of a matrix $A + E$, where $\|E\|_2 = O(\varepsilon) \|A\|_2$.

The approximate number of floating-point operations for real flavors is

$(4/3)n^3$	if $m = n$,
$(2/3)n^2(3m-n)$	if $m > n$,
$(2/3)m^2(3n-m)$	if $m < n$.

The number of operations for complex flavors is 4 times greater.

To solve a set of least-squares problems minimizing $\|Ax - b\|_2$ for all columns *b* of a given matrix *B*, you can call the following:

`?geqpf` (this routine)	to factorize $AP = QR$;
`?ormqr`	to compute $C = Q^TB$ (for real matrices);
`?unmqr`	to compute $C = Q^HB$ (for complex matrices);
`?trsm` (a BLAS routine)	to solve $RX = C$.

(The columns of the computed *X* are the permuted least-squares solution vectors *x*; the output array *`jpvt`* specifies the permutation order.)

To compute the elements of *Q* explicitly, call

`?orgqr`	(for real matrices)
`?ungqr`	(for complex matrices).

?geqp3

Computes the QR factorization of a general m by n matrix with column pivoting using Level 3 BLAS.

Syntax

```
call sgeqp3 ( m, n, a, lda, jpvt, tau, work, lwork, info )
call dgeqp3 ( m, n, a, lda, jpvt, tau, work, lwork, info )
call cgeqp3 ( m, n, a, lda, jpvt, tau, work, lwork, rwork, info )
call zgeqp3 ( m, n, a, lda, jpvt, tau, work, lwork, rwork, info )
```

Description

The routine forms the *QR* factorization of a general *m* by *n* matrix *A* with column pivoting: *AP* = *QR* (see *Orthogonal Factorizations* on page 4-5) using Level 3 BLAS. Here *P* denotes an *n* by *n* permutation matrix.
Use this routine instead of `?geqpf` for better performance.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) *elementary reflectors*. Routines are provided to work with *Q* in this representation.

Input Parameters

m	`INTEGER`. The number of rows in the matrix *A* ($m \geq 0$).
n	`INTEGER`. The number of columns in *A* ($n \geq 0$).
a, work	`REAL` for `sgeqp3` `DOUBLE PRECISION` for `dgeqp3` `COMPLEX` for `cgeqp3` `DOUBLE COMPLEX` for `zgeqp3`. Arrays: *a* (*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work* (*lwork*) is a workspace array.
lda	`INTEGER`. The first dimension of *a*; at least max(1, *m*).
lwork	`INTEGER`. The size of the *work* array; must be at least max(1, 3**n*+1) for real flavors, and at least max(1, *n*+1) for complex flavors.

jpvt INTEGER. Array, DIMENSION at least max(1, *n*).

On entry, if *jpvt*(*i*)≠ 0, the *i*th column of *A* is moved to the beginning of *AP* before the computation, and fixed in place during the computation.
If *jpvt*(*i*) = 0, the *i*th column of *A* is a free column (that is, it may be interchanged during the computation with any other free column).

rwork REAL for cgeqp3
DOUBLE PRECISION for zgeqp3.
A workspace array, DIMENSION at least max(1, 2**n*). Used in complex flavors only.

Output Parameters

a Overwritten by the factorization data as follows:

If $m \geq n$, the elements below the diagonal are overwritten by the details of the unitary (orthogonal) matrix *Q*, and the upper triangle is overwritten by the corresponding elements of the upper triangular matrix *R*.

If $m < n$, the strictly lower triangular part is overwritten by the details of the matrix *Q*, and the remaining elements are overwritten by the corresponding elements of the *m* by *n* upper trapezoidal matrix *R*.

tau REAL for sgeqp3
DOUBLE PRECISION for dgeqp3
COMPLEX for cgeqp3
DOUBLE COMPLEX for zgeqp3.
Array, DIMENSION at least max (1, min(*m*, *n*)).
Contains scalar factors of the elementary reflectors for the matrix *Q*.

jpvt Overwritten by details of the permutation matrix *P* in the factorization *AP* = *QR*. More precisely, the columns of *AP* are the columns of *A* in the following order:
jpvt(1), *jpvt*(2), ..., *jpvt*(*n*).

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

To solve a set of least-squares problems minimizing $\|Ax - b\|_2$ for all columns *b* of a given matrix *B*, you can call the following:

?geqp3 (this routine) to factorize *AP* = *QR*;

`?ormqr` to compute $C = Q^T B$ (for real matrices);

`?unmqr` to compute $C = Q^H B$ (for complex matrices);

`?trsm` (a BLAS routine) to solve $RX = C$.

(The columns of the computed X are the permuted least-squares solution vectors x; the output array *jpvt* specifies the permutation order.)

To compute the elements of Q explicitly, call

`?orgqr` (for real matrices)

`?ungqr` (for complex matrices).

?orgqr

Generates the real orthogonal matrix Q of the QR factorization formed by `?geqrf`.

Syntax

```
call sorgqr ( m, n, k, a, lda, tau, work, lwork, info )
call dorgqr ( m, n, k, a, lda, tau, work, lwork, info )
```

Description

The routine generates the whole or part of *m* by *m* orthogonal matrix *Q* of the *QR* factorization formed by the routines `sgeqrf/dgeqrf` (see page 4-7) or `sgeqpf/dgeqpf` (see page 4-9). Use this routine after a call to `sgeqrf/dgeqrf` or `sgeqpf/dgeqpf`.

Usually *Q* is determined from the *QR* factorization of an *m* by *p* matrix *A* with $m \ge p$. To compute the whole matrix *Q*, use:

```
call ?orgqr ( m, m, p, a, lda, tau, work, lwork, info )
```

To compute the leading *p* columns of *Q* (which form an orthonormal basis in the space spanned by the columns of *A*):

```
call ?orgqr ( m, p, p, a, lda, tau, work, lwork, info )
```

To compute the matrix Q^k of the *QR* factorization of *A*'s leading *k* columns:

```
call ?orgqr ( m, m, k, a, lda, tau, work, lwork, info )
```

To compute the leading *k* columns of Q^k (which form an orthonormal basis in the space spanned by *A*'s leading *k* columns):

```
call ?orgqr ( m, k, k, a, lda, tau, work, lwork, info )
```

Input Parameters

m	`INTEGER`. The order of the orthogonal matrix *Q* ($m \ge 0$).
n	`INTEGER`. The number of columns of *Q* to be computed ($0 \le n \le m$).
k	`INTEGER`. The number of elementary reflectors whose product defines the matrix *Q* ($0 \le k \le n$).

a, tau, work	`REAL` for `sorgqr` `DOUBLE PRECISION` for `dorgqr` Arrays: *a*(*lda*,*) and *tau*(*) are the arrays returned by `sgeqrf` / `dgeqrf` or `sgeqpf` / `dgeqpf`. The second dimension of *a* must be at least max(1, *n*). The dimension of *tau* must be at least max(1, *k*). *work* (*lwork*) is a workspace array.
lda	`INTEGER`. The first dimension of *a*; at least max(1, *m*).
lwork	`INTEGER`. The size of the *work* array (*lwork* $\ge$ *n*) See *Application notes* for the suggested value of *lwork*.

Output Parameters

a	Overwritten by *n* leading columns of the *m* by *m* orthogonal matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* =*n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The computed *Q* differs from an exactly orthogonal matrix by a matrix *E* such that $\|E\|_2 = O(\varepsilon)\|A\|_2$ where ε is the machine precision.

The total number of floating-point operations is approximately
$4*m*n*k - 2*(m + n)*k^2 + (4/3)*k^3$.
If $n = k$, the number is approximately $(2/3)*n^2*(3m - n)$.

The complex counterpart of this routine is ?ungqr.

?ormqr

Multiplies a real matrix by the orthogonal matrix Q of the QR factorization formed by `?geqrf` *or* `?geqpf`.

Syntax

```
call sormqr ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
call dormqr ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
```

Description

The routine multiplies a real matrix C by Q or Q^T, where Q is the orthogonal matrix Q of the QR factorization formed by the routines `sgeqrf/dgeqrf` (see page 4-7) or `sgeqpf/dgeqpf` (see page 4-9).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products QC, Q^TC, CQ, or CQ^T (overwriting the result on C).

Input Parameters

side — `CHARACTER*1`. Must be either `'L'` or `'R'`.
If *side* = `'L'`, Q or Q^T is applied to C from the left.
If *side* = `'R'`, Q or Q^T is applied to C from the right.

trans — `CHARACTER*1`. Must be either `'N'` or `'T'`.
If *trans* = `'N'`, the routine multiplies C by Q.
If *trans* = `'T'`, the routine multiplies C by Q^T.

m — `INTEGER`. The number of rows in the matrix C ($m \geq 0$).

n — `INTEGER`. The number of columns in C ($n \geq 0$).

k — `INTEGER`. The number of elementary reflectors whose product defines the matrix Q. Constraints:
$0 \leq k \leq m$ if *side* = `'L'`;
$0 \leq k \leq n$ if *side* = `'R'`.

a,work,tau,c — `REAL` for `sgeqrf`
`DOUBLE PRECISION` for `dgeqrf`.
Arrays:
a(*lda*,*) and *tau*(*) are the arrays returned by `sgeqrf` / `dgeqrf` or

sgeqpf / dgeqpf.
The second dimension of *a* must be at least max(1, *k*).
The dimension of *tau* must be at least max(1, *k*).

c(*ldc*,*) contains the matrix *C*.
The second dimension of *c* must be at least max(1, *n*)

work(*lwork*) is a workspace array.

lda — INTEGER. The first dimension of *a*. Constraints:
$lda \geq \max(1, m)$ if *side* = 'L';
$lda \geq \max(1, n)$ if *side* = 'R'.

ldc — INTEGER. The first dimension of *c*. Constraint:
$ldc \geq \max(1, m)$.

lwork — INTEGER. The size of the *work* array. Constraints:
$lwork \geq \max(1, n)$ if *side* = 'L';
$lwork \geq \max(1, m)$ if *side* = 'R'.
See *Application notes* for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product QC, Q^TC, CQ, or CQ^T (as specified by *side* and *trans*).

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* = *n* * *blocksize* (if *side* = 'L') or *lwork* = *m* * *blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The complex counterpart of this routine is ?unmqr.

?ungqr

Generates the complex unitary matrix Q of the QR factorization formed by `?geqrf`.

Syntax

```
call cungqr ( m, n, k, a, lda, tau, work, lwork, info )
call zungqr ( m, n, k, a, lda, tau, work, lwork, info )
```

Description

The routine generates the whole or part of *m* by *m* unitary matrix *Q* of the *QR* factorization formed by the routines `cgeqrf/zgeqrf` (see page 4-7) or `cgeqpf/zgeqpf` (see page 4-9). Use this routine after a call to `cgeqrf/zgeqrf` or `cgeqpf/zgeqpf`.

Usually *Q* is determined from the *QR* factorization of an *m* by *p* matrix *A* with $m \geq p$. To compute the whole matrix *Q*, use:

```
call ?ungqr ( m, m, p, a, lda, tau, work, lwork, info )
```

To compute the leading *p* columns of *Q* (which form an orthonormal basis in the space spanned by the columns of *A*):

```
call ?ungqr ( m, p, p, a, lda, tau, work, lwork, info )
```

To compute the matrix Q^k of the *QR* factorization of *A*'s leading *k* columns:

```
call ?ungqr ( m, m, k, a, lda, tau, work, lwork, info )
```

To compute the leading *k* columns of Q^k (which form an orthonormal basis in the space spanned by *A*'s leading *k* columns):

```
call ?ungqr ( m, k, k, a, lda, tau, work, lwork, info )
```

Input Parameters

m	`INTEGER`. The order of the unitary matrix *Q* ($m \geq 0$).
n	`INTEGER`. The number of columns of *Q* to be computed ($0 \leq n \leq m$).
k	`INTEGER`. The number of elementary reflectors whose product defines the matrix *Q* ($0 \leq k \leq n$).

a, tau, work	COMPLEX for cungqr DOUBLE COMPLEX for zungqr Arrays: *a(lda,*)* and *tau(*)* are the arrays returned by cgeqrf/zgeqrf or cgeqpf/zgeqpf. The second dimension of *a* must be at least max(1, *n*). The dimension of *tau* must be at least max(1, *k*). *work* (*lwork*) is a workspace array.
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array (*lwork* ≥ *n*) See *Application notes* for the suggested value of *lwork*.

Output Parameters

a	Overwritten by *n* leading columns of the *m* by *m* unitary matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* =*n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The computed Q differs from an exactly unitary matrix by a matrix E such that $\|E\|_2 = O(\varepsilon)\|A\|_2$ where ε is the machine precision.

The total number of floating-point operations is approximately
$16*m*n*k - 8*(m + n)*k^2 + (16/3)*k^3$.
If $n = k$, the number is approximately $(8/3)*n^2*(3m - n)$.

The real counterpart of this routine is ?orgqr.

?unmqr

Multiplies a complex matrix by the unitary matrix Q of the QR factorization formed by `?geqrf`.

Syntax

```
call cunmqr ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
call zunmqr ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
```

Description

The routine multiplies a rectangular complex matrix C by Q or Q^H, where Q is the unitary matrix Q of the QR factorization formed by the routines `cgeqrf`/`zgeqrf` (see page 4-7) or `cgeqpf`/`zgeqpf` (see page 4-9).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products QC, Q^HC, CQ, or CQ^H (overwriting the result on C).

Input Parameters

side — `CHARACTER*1`. Must be either `'L'` or `'R'`.
If *side* = `'L'`, Q or Q^H is applied to C from the left.
If *side* = `'R'`, Q or Q^H is applied to C from the right.

trans — `CHARACTER*1`. Must be either `'N'` or `'C'`.
If *trans* = `'N'`, the routine multiplies C by Q.
If *trans* = `'C'`, the routine multiplies C by Q^H.

m — `INTEGER`. The number of rows in the matrix C ($m \geq 0$).

n — `INTEGER`. The number of columns in C ($n \geq 0$).

k — `INTEGER`. The number of elementary reflectors whose product defines the matrix Q. Constraints:
$0 \leq k \leq m$ if *side* = `'L'`;
$0 \leq k \leq n$ if *side* = `'R'`.

a, work, tau, c — `COMPLEX` for `cgeqrf`
`DOUBLE COMPLEX` for `zgeqrf`.
Arrays:
a(*lda*,*) and *tau*(*) are the arrays returned by `cgeqrf` / `zgeqrf` or

`cgeqpf` / `zgeqpf`.
The second dimension of *a* must be at least max(1, *k*).
The dimension of *tau* must be at least max(1, *k*).

c(*ldc*,*) contains the matrix *C*.
The second dimension of *c* must be at least max(1, *n*)

work(*lwork*) is a workspace array.

lda — `INTEGER`. The first dimension of *a*. Constraints:
lda ≥ max(1, *m*) if *side* = `'L'`;
lda ≥ max(1, *n*) if *side* = `'R'`.

ldc — `INTEGER`. The first dimension of *c*. Constraint:
ldc ≥ max(1, *m*).

lwork — `INTEGER`. The size of the *work* array. Constraints:
lwork ≥ max(1, *n*) if *side* = `'L'`;
lwork ≥ max(1, *m*) if *side* = `'R'`.
See *Application notes* for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product QC, Q^HC, CQ, or CQ^H (as specified by *side* and *trans*).

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* = *n***blocksize* (if *side* = `'L'`) or *lwork* = *m***blocksize* (if *side* = `'R'`) where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The real counterpart of this routine is `?ormqr`.

?gelqf

Computes the LQ factorization of a general m by n matrix.

Syntax

```
call sgelqf ( m, n, a, lda, tau, work, lwork, info )
call dgelqf ( m, n, a, lda, tau, work, lwork, info )
call cgelqf ( m, n, a, lda, tau, work, lwork, info )
call zgelqf ( m, n, a, lda, tau, work, lwork, info )
```

Description

The routine forms the *LQ* factorization of a general *m* by *n* matrix *A* (see *Orthogonal Factorizations* on page 4-5). No pivoting is performed.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) *elementary reflectors*. Routines are provided to work with *Q* in this representation.

Input Parameters

m	INTEGER. The number of rows in the matrix *A* ($m \geq 0$).
n	INTEGER. The number of columns in *A* ($n \geq 0$).
a, work	REAL for sgelqf DOUBLE PRECISION for dgelqf COMPLEX for cgelqf DOUBLE COMPLEX for zgelqf. Arrays: *a*(*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work*(*lwork*) is a workspace array.
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array; at least max(1, *m*). See *Application notes* for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the factorization data as follows: If $m \le n$, the elements above the diagonal are overwritten by the details of the unitary (orthogonal) matrix *Q*, and the lower triangle is overwritten by the corresponding elements of the lower triangular matrix *L*. If $m > n$, the strictly upper triangular part is overwritten by the details of the matrix *Q*, and the remaining elements are overwritten by the corresponding elements of the *m* by *n* lower trapezoidal matrix *L*.
tau	`REAL` for `sgelqf` `DOUBLE PRECISION` for `dgelqf` `COMPLEX` for `cgelqf` `DOUBLE COMPLEX` for `zgelqf`. Array, `DIMENSION` at least max(1, min(*m*, *n*)). Contains additional information on the matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* =*m** *blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The computed factorization is the exact factorization of a matrix $A + E$, where $\|E\|_2 = O(\varepsilon)$ $\|A\|_2$.

The approximate number of floating-point operations for real flavors is

$(4/3)n^3$	if $m = n$,
$(2/3)n^2(3m-n)$	if $m > n$,
$(2/3)m^2(3n-m)$	if $m < n$.

The number of operations for complex flavors is 4 times greater.

To find the minimum-norm solution of an underdetermined least-squares problem minimizing $\|Ax - b\|_2$ for all columns b of a given matrix B, you can call the following:

`?gelqf` (this routine)	to factorize $A = LQ$;
`?trsm` (a BLAS routine)	to solve $LY = B$ for Y;
`?ormlq`	to compute $X = (Q_1)^T Y$ (for real matrices);
`?unmlq`	to compute $X = (Q_1)^H Y$ (for complex matrices).

(The columns of the computed X are the minimum-norm solution vectors x. Here A is an m by n matrix with $m < n$; Q_1 denotes the first m columns of Q).

To compute the elements of Q explicitly, call

`?orglq`	(for real matrices)
`?unglq`	(for complex matrices).

?orglq

Generates the real orthogonal matrix Q of the LQ factorization formed by `?gelqf`.

Syntax

```
call sorglq ( m, n, k, a, lda, tau, work, lwork, info )
call dorglq ( m, n, k, a, lda, tau, work, lwork, info )
```

Description

The routine generates the whole or part of n by n orthogonal matrix Q of the LQ factorization formed by the routines `sgelqf/dgelqf` (see page 4-23). Use this routine after a call to `sgelqf/dgelqf`.

Usually Q is determined from the LQ factorization of an p by n matrix A with $n \geq p$. To compute the whole matrix Q, use:

```
call ?orglq ( n, n, p, a, lda, tau, work, lwork, info )
```

To compute the leading p rows of Q (which form an orthonormal basis in the space spanned by the rows of A):

```
call ?orglq ( p, n, p, a, lda, tau, work, lwork, info )
```

To compute the matrix Q^k of the LQ factorization of A's leading k rows:

```
call ?orglq ( n, n, k, a, lda, tau, work, lwork, info )
```

To compute the leading k rows of Q^k (which form an orthonormal basis in the space spanned by A's leading k rows):

```
call ?orgqr ( k, n, k, a, lda, tau, work, lwork, info )
```

Input Parameters

m	`INTEGER`. The number of rows of Q to be computed ($0 \leq m \leq n$).
n	`INTEGER`. The order of the orthogonal matrix Q ($n \geq m$).
k	`INTEGER`. The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq m$).

a, tau, work	REAL for sorglq DOUBLE PRECISION for dorglq Arrays: *a*(*lda*,*) and *tau*(*) are the arrays returned by sgelqf/dgelqf. The second dimension of *a* must be at least max(1, *n*). The dimension of *tau* must be at least max(1, *k*). *work*(*lwork*) is a workspace array.
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array; at least max(1, *m*). See *Application notes* for the suggested value of *lwork*.

Output Parameters

a	Overwritten by *m* leading rows of the *n* by *n* orthogonal matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* =*m***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The computed *Q* differs from an exactly orthogonal matrix by a matrix *E* such that $||E||_2 = O(\varepsilon) ||A||_2$ where ε is the machine precision.

The total number of floating-point operations is approximately
$4*m*n*k - 2*(m + n)*k^2 + (4/3)*k^3$.
If $m = k$, the number is approximately $(2/3)*m^2*(3n - m)$.

The complex counterpart of this routine is ?unglq.

?ormlq

Multiplies a real matrix by the orthogonal matrix Q of the LQ factorization formed by `?gelqf`.

Syntax

```
call sormlq ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
call dormlq ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
```

Description

The routine multiplies a real *m*-by-*n* matrix *C* by *Q* or Q^T, where *Q* is the orthogonal matrix *Q* of the *LQ* factorization formed by the routine `sgelqf/dgelqf` (see page 4-23).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products QC, Q^TC, CQ, or CQ^T (overwriting the result on *C*).

Input Parameters

side	`CHARACTER*1`. Must be either `'L'` or `'R'`. If *side* = `'L'`, *Q* or Q^T is applied to *C* from the left. If *side* = `'R'`, *Q* or Q^T is applied to *C* from the right.
trans	`CHARACTER*1`. Must be either `'N'` or `'T'`. If *trans* = `'N'`, the routine multiplies *C* by *Q*. If *trans* = `'T'`, the routine multiplies *C* by Q^T.
m	`INTEGER`. The number of rows in the matrix *C* ($m \ge 0$).
n	`INTEGER`. The number of columns in *C* ($n \ge 0$).
k	`INTEGER`. The number of elementary reflectors whose product defines the matrix *Q*. Constraints: $0 \le k \le m$ if *side* = `'L'`; $0 \le k \le n$ if *side* = `'R'`.
a,work,tau,c	`REAL` for `sormlq` `DOUBLE PRECISION` for `dormlq`. Arrays: *a*(*lda*,*) and *tau*(*) are arrays returned by `?gelqf`. The second dimension of *a* must be:

at least max(1, *m*) if *side* ='L';
at least max(1, *n*) if *side* ='R'.
The dimension of *tau* must be at least max(1, *k*).

c(*ldc*,*) contains the matrix *C*.
The second dimension of *c* must be at least max(1, *n*)

work(*lwork*) is a workspace array.

lda — INTEGER. The first dimension of *a*; *lda* ≥ max(1, *k*).

ldc — INTEGER. The first dimension of *c*; *ldc* ≥ max(1, *m*).

lwork — INTEGER. The size of the *work* array. Constraints:
lwork ≥ max(1, *n*) if *side* ='L';
lwork ≥ max(1, *m*) if *side* ='R'.
See *Application notes* for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product *QC*, Q^TC, *CQ*, or CQ^T (as specified by *side* and *trans*).

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* = *n***blocksize* (if *side* ='L') or *lwork* = *m***blocksize* (if *side* ='R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The complex counterpart of this routine is ?unmlq.

?unglq

Generates the complex unitary matrix Q of the LQ factorization formed by `?gelqf`.

Syntax

```
call cunglq ( m, n, k, a, lda, tau, work, lwork, info )
call zunglq ( m, n, k, a, lda, tau, work, lwork, info )
```

Description

The routine generates the whole or part of n by n unitary matrix Q of the LQ factorization formed by the routines `cgelqf`/`zgelqf` (see page 4-23). Use this routine after a call to `cgelqf`/`zgelqf`.

Usually Q is determined from the LQ factorization of an p by n matrix A with $n \geq p$. To compute the whole matrix Q, use:

```
call ?unglq ( n, n, p, a, lda, tau, work, lwork, info )
```

To compute the leading p rows of Q (which form an orthonormal basis in the space spanned by the rows of A):

```
call ?unglq ( p, n, p, a, lda, tau, work, lwork, info )
```

To compute the matrix Q^k of the LQ factorization of A's leading k rows:

```
call ?unglq ( n, n, k, a, lda, tau, work, lwork, info )
```

To compute the leading k rows of Q^k (which form an orthonormal basis in the space spanned by A's leading k rows):

```
call ?ungqr ( k, n, k, a, lda, tau, work, lwork, info )
```

Input Parameters

m	`INTEGER`. The number of rows of Q to be computed ($0 \leq m \leq n$).
n	`INTEGER`. The order of the unitary matrix Q ($n \geq m$).
k	`INTEGER`. The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq m$).

a, tau, work	`COMPLEX` for `cunglq` `DOUBLE COMPLEX` for `zunglq` Arrays: *a(lda,*)* and *tau(*)* are the arrays returned by `sgelqf/dgelqf`. The second dimension of *a* must be at least max(1, *n*). The dimension of *tau* must be at least max(1, *k*). *work* (*lwork*) is a workspace array.
lda	`INTEGER`. The first dimension of *a*; at least max(1, *m*).
lwork	`INTEGER`. The size of the *work* array; at least max(1, *m*). See *Application notes* for the suggested value of *lwork*.

Output Parameters

a	Overwritten by *m* leading rows of the *n* by *n* unitary matrix *Q*.
work(1)	If *info* = 0, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* = *m***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work(1)* and use this value for subsequent runs.

The computed Q differs from an exactly unitary matrix by a matrix E such that $||E||_2 = O(\varepsilon)\ ||A||_2$ where ε is the machine precision.

The total number of floating-point operations is approximately
$16*m*n*k - 8*(m+n)*k^2 + (16/3)*k^3$.
If $m = k$, the number is approximately $(8/3)*m^2*(3n - m)$.

The real counterpart of this routine is ?orglq.

?unmlq

Multiplies a complex matrix by the unitary matrix Q of the LQ factorization formed by `?gelqf`.

Syntax

```
call cunmlq ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
call zunmlq ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
```

Description

The routine multiplies a real m-by-n matrix C by Q or Q^H, where Q is the unitary matrix Q of the LQ factorization formed by the routine `cgelqf`/`zgelqf` (see page 4-23).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products QC, Q^HC, CQ, or CQ^H (overwriting the result on C).

Input Parameters

side	`CHARACTER*1`. Must be either `'L'` or `'R'`. If *side* = `'L'`, Q or Q^H is applied to C from the left. If *side* = `'R'`, Q or Q^H is applied to C from the right.
trans	`CHARACTER*1`. Must be either `'N'` or `'C'`. If *trans* = `'N'`, the routine multiplies C by Q. If *trans* = `'C'`, the routine multiplies C by Q^H.
m	`INTEGER`. The number of rows in the matrix C ($m \ge 0$).
n	`INTEGER`. The number of columns in C ($n \ge 0$).
k	`INTEGER`. The number of elementary reflectors whose product defines the matrix Q. Constraints: $0 \le k \le m$ if *side* = `'L'`; $0 \le k \le n$ if *side* = `'R'`.
a,work,tau,c	`COMPLEX` for `cunmlq` `DOUBLE COMPLEX` for `zunmlq`. Arrays: *a*(*lda*,*) and *tau*(*) are arrays returned by `?gelqf`. The second dimension of *a* must be:

at least max(1, *m*) if *side* ='L';
at least max(1, *n*) if *side* ='R'.
The dimension of *tau* must be at least max(1, *k*).

c(*ldc*,*) contains the matrix *C*.
The second dimension of *c* must be at least max(1, *n*)

work(*lwork*) is a workspace array.

lda — INTEGER. The first dimension of *a*; *lda* ≥ max(1, *k*).

ldc — INTEGER. The first dimension of *c*; *ldc* ≥ max(1, *m*).

lwork — INTEGER. The size of the *work* array. Constraints:
lwork ≥ max(1, *n*) if *side* ='L';
lwork ≥ max(1, *m*) if *side* ='R'.
See *Application notes* for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product QC, Q^HC, CQ, or CQ^H (as specified by *side* and *trans*).

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* = *n* * *blocksize* (if *side* ='L') or *lwork* = *m* * *blocksize* (if *side* ='R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The real counterpart of this routine is ?ormlq.

?geqlf

Computes the QL factorization of a general m by n matrix.

Syntax

```
call sgeqlf ( m, n, a, lda, tau, work, lwork, info )
call dgeqlf ( m, n, a, lda, tau, work, lwork, info )
call cgeqlf ( m, n, a, lda, tau, work, lwork, info )
call zgeqlf ( m, n, a, lda, tau, work, lwork, info )
```

Description

The routine forms the *QL* factorization of a general `m`-by-`n` matrix *A*.
No pivoting is performed.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(`m`, `n`) *elementary reflectors*. Routines are provided to work with *Q* in this representation.

Input Parameters

m	`INTEGER`. The number of rows in the matrix *A* ($m \ge 0$).
n	`INTEGER`. The number of columns in *A* ($n \ge 0$).
a, work	`REAL` for `sgeqlf` `DOUBLE PRECISION` for `dgeqlf` `COMPLEX` for `cgeqlf` `DOUBLE COMPLEX` for `zgeqlf`. Arrays: `a(lda,*)` contains the matrix *A*. The second dimension of `a` must be at least max(1, `n`). `work(lwork)` is a workspace array.
lda	`INTEGER`. The first dimension of `a`; at least max(1, `m`).
lwork	`INTEGER`. The size of the `work` array; at least max(1, `n`). See *Application notes* for the suggested value of `lwork`.

Output Parameters

a	Overwritten on exit by the factorization data as follows: if $m \geq n$, the lower triangle of the subarray *a*(*m*-*n*+1:*m*, 1:*n*) contains the *n*-by-*n* lower triangular matrix *L*; if $m \leq n$, the elements on and below the (*n*-*m*)th superdiagonal contain the *m*-by-*n* lower trapezoidal matrix *L;* in both cases, the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.
tau	`REAL` for `sgeqlf` `DOUBLE PRECISION` for `dgeqlf` `COMPLEX` for `cgeqlf` `DOUBLE COMPLEX` for `zgeqlf`. Array, `DIMENSION` at least max(1, min(*m*, *n*)). Contains scalar factors of the elementary reflectors for the matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* =*n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.
Related routines include:

?orgql	to generate matrix Q (for real matrices);
?ungql	to generate matrix Q (for complex matrices);
?ormql	to apply matrix Q (for real matrices);
?unmql	to apply matrix Q (for complex matrices).

?orgql

Generates the real matrix Q of the QL factorization formed by `?geqlf`.

Syntax

```
call sorgql ( m, n, k, a, lda, tau, work, lwork, info )
call dorgql ( m, n, k, a, lda, tau, work, lwork, info )
```

Description

The routine generates an *m*-by-*n* real matrix Q with orthonormal columns, which is defined as the last *n* columns of a product of *k* elementary reflectors H_i of order *m* : $Q = H_k \cdots H_2H_1$ as returned by the routines sgeqlf/dgeqlf . Use this routine after a call to sgeqlf/dgeqlf.

Input Parameters

m — INTEGER. The number of rows of the matrix Q ($m \ge 0$).

n — INTEGER. The number of columns of the matrix Q ($m \ge n \ge 0$).

k — INTEGER. The number of elementary reflectors whose product defines the matrix Q ($n \ge k \ge 0$).

a, tau, work — REAL for sorgql
DOUBLE PRECISION for dorgql
Arrays: *a*(*lda*,*), *tau*(*), *work*(*lwork*).

On entry, the ($n - k + i$)th column of *a* must contain the vector which defines the elementary reflector H_i, for i = 1,2,...,*k*, as returned by sgeqlf/dgeqlf in the last *k* columns of its array argument *a*;
tau(i) must contain the scalar factor of the elementary reflector H_i, as returned by sgeqlf/dgeqlf;

The second dimension of *a* must be at least max(1, *n*).
The dimension of *tau* must be at least max(1, *k*).

work(*lwork*) is a workspace array.

lda — INTEGER. The first dimension of *a*; at least max(1, *m*).

lwork INTEGER. The size of the *work* array; at least max(1, *n*).
See *Application notes* for the suggested value of *lwork*.

Output Parameters

a Overwritten by the *m*-by-*n* matrix *Q*.

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* =*n** *blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The complex counterpart of this routine is ?ungql.

?ungql

Generates the complex matrix Q of the QL factorization formed by `?geqlf`.

Syntax

```
call cungql ( m, n, k, a, lda, tau, work, lwork, info )
call zungql ( m, n, k, a, lda, tau, work, lwork, info )
```

Description

The routine generates an *m*-by-*n* complex matrix Q with orthonormal columns, which is defined as the last *n* columns of a product of *k* elementary reflectors H_i of order *m* : $Q = H_k \cdots H_2 H_1$ as returned by the routines cgeqlf/zgeqlf . Use this routine after a call to `cgeqlf/zgeqlf`.

Input Parameters

m — `INTEGER`. The number of rows of the matrix Q ($m \geq 0$).

n — `INTEGER`. The number of columns of the matrix Q ($m \geq n \geq 0$).

k — `INTEGER`. The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).

a, tau, work — `COMPLEX` for `cungql`
`DOUBLE COMPLEX` for `zungql`
Arrays: *a*(*lda*,*), *tau*(*), *work*(*lwork*).

On entry, the (*n* - *k* + *i*)th column of *a* must contain the vector which defines the elementary reflector H_i, for i = 1,2,...,*k*, as returned by `cgeqlf/zgeqlf` in the last *k* columns of its array argument *a*;
tau(i) must contain the scalar factor of the elementary reflector H_i, as returned by `cgeqlf/zgeqlf`;

The second dimension of *a* must be at least max(1, *n*).
The dimension of *tau* must be at least max(1, *k*).

work(*lwork*) is a workspace array.

lda — `INTEGER`. The first dimension of *a*; at least max(1, *m*).

lwork — INTEGER. The size of the *work* array; at least max(1, *n*). See *Application notes* for the suggested value of *lwork*.

Output Parameters

a — Overwritten by the *m*-by-*n* matrix *Q*.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* =*n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The real counterpart of this routine is ?orgql.

?ormql

Multiplies a real matrix by the orthogonal matrix Q of the QL factorization formed by `?geqlf`.

Syntax

```
call sormql ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
call dormql ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
```

Description

This routine multiplies a real m-by-n matrix C by Q or Q^T, where Q is the orthogonal matrix Q of the QL factorization formed by the routine `sgeqlf/dgeqlf` .

Depending on the parameters *side* and *trans*, the routine `?ormql` can form one of the matrix products QC, Q^TC, CQ, or CQ^T (overwriting the result over C).

Input Parameters

side	`CHARACTER*1`. Must be either `'L'` or `'R'`. If *side* =`'L'`, Q or Q^T is applied to C from the left. If *side* =`'R'`, Q or Q^T is applied to C from the right.
trans	`CHARACTER*1`. Must be either `'N'` or `'T'`. If *trans* =`'N'`, the routine multiplies C by Q. If *trans* =`'T'`, the routine multiplies C by Q^T.
m	`INTEGER`. The number of rows in the matrix C ($m \geq 0$).
n	`INTEGER`. The number of columns in C ($n \geq 0$).
k	`INTEGER`. The number of elementary reflectors whose product defines the matrix Q. Constraints: $0 \leq k \leq m$ if *side* =`'L'`; $0 \leq k \leq n$ if *side* =`'R'`.
a, tau, c, work	`REAL` for `sormql` `DOUBLE PRECISION` for `dormql`. Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*), *work*(*lwork*).

On entry, the *i*th column of *a* must contain the vector which defines the elementary reflector H_i, for i = 1,2,...,*k*, as returned by sgeqlf/dgeqlf in the last *k* columns of its array argument *a*.
The second dimension of *a* must be at least max(1, *k*).

tau(i) must contain the scalar factor of the elementary reflector H_i, as returned by sgeqlf/dgeqlf.
The dimension of *tau* must be at least max(1, *k*).

c(*ldc*,*) contains the *m*-by-*n* matrix *C*.
The second dimension of *c* must be at least max(1, *n*)

work(*lwork*) is a workspace array.

lda INTEGER. The first dimension of *a*;

if *side* ='L', $lda \geq \max(1, m)$;
if *side* ='R', $lda \geq \max(1, n)$.

ldc INTEGER. The first dimension of *c*; $ldc \geq \max(1, m)$.

lwork INTEGER. The size of the *work* array. Constraints:
$lwork \geq \max(1, n)$ if *side* ='L';
$lwork \geq \max(1, m)$ if *side* ='R'.
See *Application notes* for the suggested value of *lwork*.

Output Parameters

c Overwritten by the product QC, Q^TC, CQ, or CQ^T (as specified by *side* and *trans*).

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* = *n* * *blocksize* (if *side* ='L') or *lwork* = *m* * *blocksize* (if *side* ='R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The complex counterpart of this routine is ?unmql.

?unmql

Multiplies a complex matrix by the unitary matrix Q of the QL factorization formed by `?geqlf`.

Syntax

```
call cunmql ( side,trans,m,n,k,a, lda,tau,c, ldc,work, lwork,info )
call zunmql ( side,trans,m,n,k,a, lda,tau,c, ldc,work, lwork,info )
```

Description

The routine multiplies a complex m-by-n matrix C by Q or Q^H, where Q is the unitary matrix Q of the QL factorization formed by the routine `cgeqlf/zgeqlf` .

Depending on the parameters *side* and *trans*, the routine `?unmql` can form one of the matrix products QC, Q^HC, CQ, or CQ^H (overwriting the result over C).

Input Parameters

side — `CHARACTER*1`. Must be either `'L'` or `'R'`.
If *side* =`'L'`, Q or Q^H is applied to C from the left.
If *side* =`'R'`, Q or Q^H is applied to C from the right.

trans — `CHARACTER*1`. Must be either `'N'` or `'C'`.
If *trans* =`'N'`, the routine multiplies C by Q.
If *trans* =`'C'`, the routine multiplies C by Q^H.

m — `INTEGER`. The number of rows in the matrix C ($m \geq 0$).

n — `INTEGER`. The number of columns in C ($n \geq 0$).

k — `INTEGER`. The number of elementary reflectors whose product defines the matrix Q. Constraints:
$0 \leq k \leq m$ if *side* =`'L'`;
$0 \leq k \leq n$ if *side* =`'R'`.

a, tau, c, work — `COMPLEX` for `cunmql`
`DOUBLE COMPLEX` for `zunmql`.
Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*), *work*(*lwork*).

On entry, the `i`th column of *a* must contain the vector which defines the elementary reflector H_i, for i = 1,2,...,*k*, as returned by `cgeqlf/zgeqlf` in the last *k* columns of its array argument *a*.
The second dimension of *a* must be at least max(1, *k*).

tau(i) must contain the scalar factor of the elementary reflector H_i, as returned by `cgeqlf/zgeqlf`.
The dimension of *tau* must be at least max(1, *k*).

c(*ldc*,*) contains the *m*-by-*n* matrix *C*.
The second dimension of *c* must be at least max(1, *n*)

work(*lwork*) is a workspace array.

lda — `INTEGER`. The first dimension of *a*;

if *side* = 'L', *lda* ≥ max(1, *m*);
if *side* = 'R', *lda* ≥ max(1, *n*) .

ldc — `INTEGER`. The first dimension of *c*; *ldc* ≥ max(1, *m*).

lwork — `INTEGER`. The size of the *work* array. Constraints:
lwork ≥ max(1, *n*) if *side* = 'L';
lwork ≥ max(1, *m*) if *side* = 'R'.
See *Application notes* for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product *QC*, Q^HC, *CQ*, or CQ^H (as specified by *side* and *trans*).

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* = *n***blocksize* (if *side* = 'L') or *lwork* = *m***blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The real counterpart of this routine is `?ormql`.

?gerqf

Computes the RQ factorization of a general m by n matrix.

Syntax

```
call sgerqf ( m, n, a, lda, tau, work, lwork, info )
call dgerqf ( m, n, a, lda, tau, work, lwork, info )
call cgerqf ( m, n, a, lda, tau, work, lwork, info )
call zgerqf ( m, n, a, lda, tau, work, lwork, info )
```

Description

The routine forms the *RQ* factorization of a general `m`-by-`n` matrix *A*.
No pivoting is performed.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(`m`, `n`) *elementary reflectors*. Routines are provided to work with *Q* in this representation.

Input Parameters

m — `INTEGER`. The number of rows in the matrix *A* ($m \geq 0$).

n — `INTEGER`. The number of columns in *A* ($n \geq 0$).

a, work — `REAL` for `sgerqf`
`DOUBLE PRECISION` for `dgerqf`
`COMPLEX` for `cgerqf`
`DOUBLE COMPLEX` for `zgerqf`.
Arrays:
`a(lda,*)` contains the `m`-by-`n` matrix *A*.
The second dimension of `a` must be at least max(1, `n`).

`work(lwork)` is a workspace array.

lda — `INTEGER`. The first dimension of `a`; at least max(1, `m`).

lwork — `INTEGER`. The size of the `work` array;
`lwork` ≥ max(1, `m`).
See Application notes for the suggested value of `lwork`.

Output Parameters

a	Overwritten on exit by the factorization data as follows: if $m \le n$, the upper triangle of the subarray $a(1{:}m, n{-}m{+}1{:}n)$ contains the m-by-m upper triangular matrix R; if $m \ge n$, the elements on and above the (m-n)th subdiagonal contain the m-by-n upper trapezoidal matrix R; in both cases, the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix Q as a product of min(m,n) elementary reflectors.
tau	`REAL` for `sgerqf` `DOUBLE PRECISION` for `dgerqf` `COMPLEX` for `cgerqf` `DOUBLE COMPLEX` for `zgerqf`. Array, `DIMENSION` at least max (1, min(m, n)). Contains scalar factors of the elementary reflectors for the matrix Q.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* =m* *blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

Related routines include:

?orgrq	to generate matrix Q (for real matrices);
?ungrq	to generate matrix Q (for complex matrices);
?ormrq	to apply matrix Q (for real matrices);
?unmrq	to apply matrix Q (for complex matrices).

?orgrq

Generates the real matrix Q of the RQ factorization formed by `?gerqf`.

Syntax

```
call sorgrq ( m, n, k, a, lda, tau, work, lwork, info )
call dorgrq ( m, n, k, a, lda, tau, work, lwork, info )
```

Description

The routine generates an *m*-by-*n* real matrix *Q* with orthonormal rows, which is defined as the last *m* rows of a product of *k* elementary reflectors H_i of order *n* : $Q = H_1 H_2 \cdots H_k$ as returned by the routines sgerqf/dgerqf. Use this routine after a call to sgerqf/dgerqf.

Input Parameters

m	INTEGER. The number of rows of the matrix *Q* ($m \geq 0$).
n	INTEGER. The number of columns of the matrix *Q* ($n \geq m$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q* ($m \geq k \geq 0$).
a, tau, work	REAL for sorgrq DOUBLE PRECISION for dorgrq Arrays: *a*(*lda*,*), *tau*(*), *work*(*lwork*). On entry, the (*m* - *k* + *i*)th row of *a* must contain the vector which defines the elementary reflector H_i, for i = 1,2,...,*k*, as returned by sgerqf/dgerqf in the last *k* rows of its array argument *a*; *tau*(i) must contain the scalar factor of the elementary reflector H_i, as returned by sgerqf/dgerqf; The second dimension of *a* must be at least max(1, *n*). The dimension of *tau* must be at least max(1, *k*). *work*(*lwork*) is a workspace array.
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).

lwork INTEGER. The size of the *work* array; at least max(1, *m*).
See *Application notes* for the suggested value of *lwork*.

Output Parameters

a Overwritten by the *m*-by-*n* matrix *Q*.

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* =*m***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The complex counterpart of this routine is ?ungrq.

?ungrq

Generates the complex matrix Q of the RQ factorization formed by `?gerqf`.

Syntax

```
call cungrq ( m, n, k, a, lda, tau, work, lwork, info )
call zungrq ( m, n, k, a, lda, tau, work, lwork, info )
```

Description

The routine generates an *m*-by-*n* complex matrix Q with orthonormal rows, which is defined as the last *m* rows of a product of *k* elementary reflectors H_i of order *n*: $Q = H_1^H H_2^H \cdots H_k^H$ as returned by the routines sgerqf/dgerqf. Use this routine after a call to sgerqf/dgerqf.

Input Parameters

m — INTEGER. The number of rows of the matrix Q ($m \geq 0$).

n — INTEGER. The number of columns of the matrix Q ($n \geq m$).

k — INTEGER. The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).

a, tau, work — REAL for cungrq
DOUBLE PRECISION for zungrq
Arrays: *a*(*lda*,*), *tau*(*), *work*(*lwork*).

On entry, the ($m - k + i$)th row of *a* must contain the vector which defines the elementary reflector H_i, for i = 1,2,...,*k*, as returned by sgerqf/dgerqf in the last *k* rows of its array argument *a*;
tau(i) must contain the scalar factor of the elementary reflector H_i, as returned by sgerqf/dgerqf;

The second dimension of *a* must be at least max(1, *n*).
The dimension of *tau* must be at least max(1, *k*).

work(*lwork*) is a workspace array.

lda — INTEGER. The first dimension of *a*; at least max(1, *m*).

lwork INTEGER. The size of the *work* array; at least max(1, *m*). See *Application notes* for the suggested value of *lwork*.

Output Parameters

a Overwritten by the *m*-by-*n* matrix *Q*.

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* =*m***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The real counterpart of this routine is ?orgrq.

?ormrq

Multiplies a real matrix by the orthogonal matrix Q of the RQ factorization formed by `?gerqf`.

Syntax

```
call sormrq ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
call dormrq ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
```

Description

The routine multiplies a real *m*-by-*n* matrix *C* by *Q* or Q^T, where *Q* is the real orthogonal matrix defined as a product of *k* elementary reflectors H_i : $Q = H_1 H_2 \cdots H_k$ as returned by the *RQ* factorization routine sgerqf/dgerqf .

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products QC, Q^TC, CQ, or CQ^T (overwriting the result over *C*).

Input Parameters

side — `CHARACTER*1`. Must be either `'L'` or `'R'`.
If *side* = `'L'`, *Q* or Q^T is applied to *C* from the left.
If *side* = `'R'`, *Q* or Q^T is applied to *C* from the right.

trans — `CHARACTER*1`. Must be either `'N'` or `'T'`.
If *trans* = `'N'`, the routine multiplies *C* by *Q*.
If *trans* = `'T'`, the routine multiplies *C* by Q^T.

m — `INTEGER`. The number of rows in the matrix *C* ($m \ge 0$).

n — `INTEGER`. The number of columns in *C* ($n \ge 0$).

k — `INTEGER`. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:
$0 \le k \le m$, if *side* = `'L'`;
$0 \le k \le n$, if *side* = `'R'`.

a, tau, c, work — `REAL` for `sormrq`
`DOUBLE PRECISION` for `dormrq`.
Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*), *work*(*lwork*).

On entry, the `i`th row of `a` must contain the vector which defines the elementary reflector H_i, for i = 1,2,...,*k*, as returned by `sgerqf/dgerqf` in the last *k* rows of its array argument `a`.
The second dimension of `a` must be at least max(1, *m*) if `side` = `'L'`, and at least max(1, *n*) if `side` = `'R'`.

`tau`(i) must contain the scalar factor of the elementary reflector H_i, as returned by `sgerqf/dgerqf`.
The dimension of `tau` must be at least max(1, *k*).

`c(ldc,*)` contains the *m*-by-*n* matrix *C*.
The second dimension of `c` must be at least max(1, *n*)

`work(lwork)` is a workspace array.

`lda` — `INTEGER`. The first dimension of `a`; `lda` ≥ max(1, *k*) .

`ldc` — `INTEGER`. The first dimension of `c`; `ldc` ≥ max(1, *m*).

`lwork` — `INTEGER`. The size of the `work` array. Constraints:
`lwork` ≥ max(1, *n*) if `side` = `'L'`;
`lwork` ≥ max(1, *m*) if `side` = `'R'`.
See *Application notes* for the suggested value of `lwork`.

Output Parameters

`c` — Overwritten by the product *QC*, Q^TC, *CQ*, or CQ^T (as specified by `side` and `trans`).

`work(1)` — If `info` = 0, on exit `work(1)` contains the minimum value of `lwork` required for optimum performance. Use this `lwork` for subsequent runs.

`info` — `INTEGER`.
If `info` = 0, the execution is successful.
If `info` = `-i`, the `i`th parameter had an illegal value.

Application Notes

For better performance, try using `lwork` = `n`**blocksize* (if `side` = `'L'`) or `lwork` = `m`**blocksize* (if `side` = `'R'`) where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The complex counterpart of this routine is `?unmrq`.

?unmrq

Multiplies a complex matrix by the unitary matrix Q of the RQ factorization formed by ?gerqf.

Syntax

```
call cunmrq ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
call zunmrq ( side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info )
```

Description

The routine multiplies a complex *m*-by-*n* matrix C by Q or Q^H, where Q is the complex unitary matrix defined as a product of *k* elementary reflectors H_i : $Q = H_1^H H_2^H \cdots H_k^H$ as returned by the *RQ* factorization routine cgerqf/zgerqf .

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products QC, Q^HC, CQ, or CQ^H (overwriting the result over C).

Input Parameters

side — CHARACTER*1. Must be either 'L' or 'R'.
If *side* ='L', Q or Q^H is applied to C from the left.
If *side* ='R', Q or Q^H is applied to C from the right.

trans — CHARACTER*1. Must be either 'N' or 'C'.
If *trans* ='N', the routine multiplies C by Q.
If *trans* ='C', the routine multiplies C by Q^H.

m — INTEGER. The number of rows in the matrix C ($m \ge 0$).

n — INTEGER. The number of columns in C ($n \ge 0$).

k — INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints:
$0 \le k \le m$, if *side* ='L';
$0 \le k \le n$, if *side* ='R'.

a,tau,c,work — COMPLEX for cunmrq
DOUBLE COMPLEX for zunmrq.
Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*), *work*(*lwork*).

On entry, the `i`th row of *a* must contain the vector which defines the elementary reflector H_i, for i = 1,2,...,*k*, as returned by `cgerqf/zgerqf` in the last *k* rows of its array argument *a*.
The second dimension of *a* must be at least max(1, *m*) if *side* ='L', and at least max(1, *n*) if *side* ='R'.

tau(i) must contain the scalar factor of the elementary reflector H_i, as returned by `cgerqf/zgerqf`.
The dimension of *tau* must be at least max(1, *k*).

c(*ldc*,*) contains the *m*-by-*n* matrix *C*.
The second dimension of *c* must be at least max(1, *n*)

work(*lwork*) is a workspace array.

lda — `INTEGER`. The first dimension of *a*; *lda* ≥ max(1, *k*) .

ldc — `INTEGER`. The first dimension of *c*; *ldc* ≥ max(1, *m*).

lwork — `INTEGER`. The size of the *work* array. Constraints:
lwork ≥ max(1, *n*) if *side* ='L';
lwork ≥ max(1, *m*) if *side* ='R'.
See *Application notes* for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product *QC*, Q^HC, *CQ*, or CQ^H (as specified by *side* and *trans*).

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* = *n***blocksize* (if *side* ='L') or *lwork* = *m***blocksize* (if *side* ='R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The real counterpart of this routine is `?ormrq`.

?tzrzf

Reduces the upper trapezoidal matrix A to upper triangular form.

Syntax

```
call stzrzf ( m, n, a, lda, tau, work, lwork, info )
call dtzrzf ( m, n, a, lda, tau, work, lwork, info )
call ctzrzf ( m, n, a, lda, tau, work, lwork, info )
call ztzrzf ( m, n, a, lda, tau, work, lwork, info )
```

Description

This routine reduces the *m*-by-*n* ($m \le n$) real/complex upper trapezoidal matrix *A* to upper triangular form by means of orthogonal/unitary transformations. The upper trapezoidal matrix *A* is factored as

$$A = (\, R \;\; 0 \,) * Z,$$

where *Z* is an *n*-by-*n* orthogonal/unitary matrix and R is an *m*-by-*m* upper triangular matrix.

Input Parameters

m — `INTEGER`. The number of rows in the matrix *A* ($m \ge 0$).

n — `INTEGER`. The number of columns in *A* ($n \ge m$).

a, work — `REAL` for `stzrzf`
`DOUBLE PRECISION` for `dtzrzf`
`COMPLEX` for `ctzrzf`
`DOUBLE COMPLEX` for `ztzrzf`.
Arrays: `a(lda,*), work(lwork).`
The leading *m*-by-*n* upper trapezoidal part of the array *a* contains the matrix *A* to be factorized.
The second dimension of *a* must be at least max(1, *n*).

work is a workspace array.

lda — `INTEGER`. The first dimension of *a*; at least max(1, *m*).

lwork — `INTEGER`. The size of the *work* array;

lwork ≥ max(1, *m*).
See *Application notes* for the suggested value of *lwork*.

Output Parameters

a — Overwritten on exit by the factorization data as follows:

the leading *m*-by-*m* upper triangular part of *a* contains the upper triangular matrix *R*, and elements *m* +1 to *n* of the first *m* rows of *a*, with the array *tau*, represent the orthogonal matrix *Z* as a product of *m* elementary reflectors.

tau — REAL for stzrzf
DOUBLE PRECISION for dtzrzf
COMPLEX for ctzrzf
DOUBLE COMPLEX for ztzrzf.
Array, DIMENSION at least max (1, *m*).
Contains scalar factors of the elementary reflectors for the matrix *Z*.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* =*m** *blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.
If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run.
On exit, examine *work*(1) and use this value for subsequent runs.
Related routines include:

?ormrz — to apply matrix Q (for real matrices);

?unmrz — to apply matrix Q (for complex matrices).

?ormrz

Multiplies a real matrix by the orthogonal matrix defined from the factorization formed by `?tzrzf`.

Syntax

```
call sormrz ( side,trans,m,n,k,l,a,lda,tau,c,ldc,work,lwork,info )
call dormrz ( side,trans,m,n,k,l,a,lda,tau,c,ldc,work,lwork,info )
```

Description

The routine multiplies a real *m*-by-*n* matrix C by Q or Q^T, where Q is the real orthogonal matrix defined as a product of *k* elementary reflectors H_i : $Q = H_1 H_2 \cdots H_k$ as returned by the factorization routine stzrzf/dtzrzf .

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products QC, Q^TC, CQ, or CQ^T (overwriting the result over C).

The matrix Q is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side — CHARACTER*1. Must be either 'L' or 'R'.
If *side* = 'L', Q or Q^T is applied to C from the left.
If *side* = 'R', Q or Q^T is applied to C from the right.

trans — CHARACTER*1. Must be either 'N' or 'T'.
If *trans* = 'N', the routine multiplies C by Q.
If *trans* = 'T', the routine multiplies C by Q^T.

m — INTEGER. The number of rows in the matrix C ($m \geq 0$).

n — INTEGER. The number of columns in C ($n \geq 0$).

k — INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints:
$0 \leq k \leq m$, if *side* = 'L';
$0 \leq k \leq n$, if *side* = 'R'.

l — INTEGER.

The number of columns of the matrix *A* containing the meaningful part of the Householder reflectors. Constraints:
$0 \le l \le m$, if *side* = 'L';
$0 \le l \le n$, if *side* = 'R'.

a, tau, c, work — REAL for sormrz
DOUBLE PRECISION for dormrz.
Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*), *work*(*lwork*).

On entry, the *i*th row of *a* must contain the vector which defines the elementary reflector H_i, for i = 1,2,...,*k*, as returned by stzrzf/dtzrzf in the last *k* rows of its array argument *a*.
The second dimension of *a* must be at least max(1, *m*) if *side* = 'L', and at least max(1, *n*) if *side* = 'R'.

tau(i) must contain the scalar factor of the elementary reflector H_i, as returned by stzrzf/dtzrzf.
The dimension of *tau* must be at least max(1, *k*).

c(*ldc*,*) contains the *m*-by-*n* matrix *C*.
The second dimension of *c* must be at least max(1, *n*)

work(*lwork*) is a workspace array.

lda — INTEGER. The first dimension of *a*; $lda \ge \max(1, k)$.

ldc — INTEGER. The first dimension of *c*; $ldc \ge \max(1, m)$.

lwork — INTEGER. The size of the *work* array. Constraints:
$lwork \ge \max(1, n)$ if *side* = 'L';
$lwork \ge \max(1, m)$ if *side* = 'R'.
See *Application notes* for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product *QC*, Q^TC, *CQ*, or CQ^T (as specified by *side* and *trans*).

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using `lwork` = `n`* *blocksize* (if `side` = 'L') or `lwork` = `m`* *blocksize* (if `side` = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The complex counterpart of this routine is `?unmrz`.

?unmrz

Multiplies a complex matrix by the unitary matrix defined from the factorization formed by `?tzrzf`.

Syntax

```
call cunmrz ( side,trans,m,n,k,l,a,lda,tau,c,ldc,work,lwork,info )
call zunmrz ( side,trans,m,n,k,l,a,lda,tau,c,ldc,work,lwork,info )
```

Description

The routine multiplies a complex *m*-by-*n* matrix *C* by *Q* or Q^H, where *Q* is the unitary matrix defined as a product of *k* elementary reflectors H_i :
$Q = H_1^H H_2^H \cdots H_k^H$ as returned by the factorization routine ctzrzf/ztzrzf .

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products QC, Q^HC, CQ, or CQ^H (overwriting the result over *C*).

The matrix *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side — CHARACTER*1. Must be either 'L' or 'R'.
If *side* ='L', *Q* or Q^H is applied to *C* from the left.
If *side* ='R', *Q* or Q^H is applied to *C* from the right.

trans — CHARACTER*1. Must be either 'N' or 'C'.
If *trans* ='N', the routine multiplies *C* by *Q*.
If *trans* ='C', the routine multiplies *C* by Q^H.

m — INTEGER. The number of rows in the matrix *C* ($m \ge 0$).

n — INTEGER. The number of columns in *C* ($n \ge 0$).

k — INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:
$0 \le k \le m$, if *side* ='L';
$0 \le k \le n$, if *side* ='R'.

l — INTEGER.

The number of columns of the matrix A containing the meaningful part of the Householder reflectors. Constraints:
$0 \le l \le m$, if *side* = 'L';
$0 \le l \le n$, if *side* = 'R'.

a, *tau*, *c*, *work* — COMPLEX for cunmrz
DOUBLE COMPLEX for zunmrz.
Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*), *work*(*lwork*).

On entry, the *i*th row of *a* must contain the vector which defines the elementary reflector H_i, for i = 1,2,...,*k*, as returned by ctzrzf/ztzrzf in the last *k* rows of its array argument *a*.
The second dimension of *a* must be at least max(1, *m*) if *side* = 'L', and at least max(1, *n*) if *side* = 'R'.

tau(i) must contain the scalar factor of the elementary reflector H_i, as returned by ctzrzf/ztzrzf.
The dimension of *tau* must be at least max(1, *k*).

c(*ldc*,*) contains the *m*-by-*n* matrix *C*.
The second dimension of *c* must be at least max(1, *n*)

work(*lwork*) is a workspace array.

lda — INTEGER. The first dimension of *a*; *lda* ≥ max(1, *k*) .

ldc — INTEGER. The first dimension of *c*; *ldc* ≥ max(1, *m*).

lwork — INTEGER. The size of the *work* array. Constraints:
lwork ≥ max(1, *n*) if *side* = 'L';
lwork ≥ max(1, *m*) if *side* = 'R'.
See *Application notes* for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product QC, Q^HC, CQ, or CQ^H (as specified by *side* and *trans*).

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using `lwork` = `n`* *blocksize* (if `side` ='L') or `lwork` = `m`* *blocksize* (if `side` ='R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The real counterpart of this routine is `?ormrz`.

?ggqrf

Computes the generalized QR factorization of two matrices.

Syntax

```
call sggqrf (n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call dggqrf (n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call cggqrf (n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call zggqrf (n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
```

Description

The routine forms the generalized QR factorization of an n-by-m matrix A and an n-by-p matrix B as $A = QR$, $B = QTZ$,
where Q is an n-by-n orthogonal/unitary matrix, Z is a p-by-p orthogonal/unitary matrix, and R and T assume one of the forms:

$$R = \begin{array}{c} \\ m \\ n-m \end{array} \begin{array}{c} m \\ \begin{pmatrix} R_{11} \\ 0 \end{pmatrix} \end{array}, \quad \text{if } n \ge m$$

or

$$R = n \begin{array}{c} \begin{array}{cc} n & m-n \end{array} \\ \begin{pmatrix} R_{11} & R_{12} \end{pmatrix} \end{array}, \quad \text{if } n < m,$$

where R_{11} is upper triangular, and

$$T = n \begin{array}{c} \begin{array}{cc} p-n & n \end{array} \\ \begin{pmatrix} 0 & T_{12} \end{pmatrix} \end{array}, \quad \text{if } n \le p, \text{ or}$$

$$T = \begin{array}{c} \\ n-p \\ p \end{array} \begin{array}{c} p \\ \begin{pmatrix} T_{11} \\ T_{21} \end{pmatrix} \end{array}, \quad \text{if } n > p$$

where T_{12} or T_{21} is a *p*-by-*p* upper triangular matrix.

In particular, if *B* is square and nonsingular, the *GQR* factorization of *A* and *B* implicitly gives the *QR* factorization of $B^{-1}A$ as:

$$B^{-1}A = Z^H(T^{-1}R)$$

Input Parameters

n	INTEGER. The number of rows of the matrices *A* and *B* ($n \geq 0$).
m	INTEGER. The number of columns in *A* ($m \geq 0$).
p	INTEGER. The number of columns in *B* ($p \geq 0$).
a, b, work	REAL for sggqrf DOUBLE PRECISION for dggqrf COMPLEX for cggqrf DOUBLE COMPLEX for zggqrf. Arrays: a(*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *m*). b(*ldb*,*) contains the matrix *B*. The second dimension of *b* must be at least max(1, *p*). *work*(*lwork*) is a workspace array.
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
ldb	INTEGER. The first dimension of *b*; at least max(1, *n*).
lwork	INTEGER. The size of the *work* array; must be at least max(1, *n*, *m*, *p*) See *Application notes* for the suggested value of *lwork*.

Output Parameters

a, b	Overwritten by the factorization data as follows: on exit, the elements on and above the diagonal of the array *a* contain the min(*n*,*m*)-by-*m* upper trapezoidal matrix *R* (*R* is upper triangular if $n \geq m$); the elements below the diagonal, with the array *taua*, represent the orthogonal/unitary matrix *Q* as a product of min(*n*,*m*) elementary reflectors ;

if $n \le p$, the upper triangle of the subarray $b(1:n, p-n+1:p)$ contains the n-by-n upper triangular matrix T;
if $n > p$, the elements on and above the $(n-p)$th subdiagonal contain the n-by-p upper trapezoidal matrix T; the remaining elements, with the array *taub*, represent the orthogonal/unitary matrix Z as a product of elementary reflectors.

taua, taub — `REAL` for `sggqrf`
`DOUBLE PRECISION` for `dggqrf`
`COMPLEX` for `cggqrf`
`DOUBLE COMPLEX` for `zggqrf`.
Arrays, `DIMENSION` at least max (1, min(n, m)) for *taua* and at least max (1, min(n, p)) for *taub*.
The array *taua* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Q.

The array *taub* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Z.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using
lwork $\ge$ max(n,m,p)*max($nb1$,$nb2$,$nb3$),
where $nb1$ is the optimal blocksize for the QR factorization of an n-by-m matrix, $nb2$ is the optimal blocksize for the RQ factorization of an n-by-p matrix, and $nb3$ is the optimal blocksize for a call of `?ormqr/?unmqr`.

?ggrqf

Computes the generalized RQ factorization of two matrices.

Syntax

```
call sggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call dggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call cggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call zggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
```

Description

The routine forms the generalized *RQ* factorization of an m-by-n matrix A and an p-by-n matrix B as $A = RQ$, $B = ZTQ$,
where Q is an n-by-n orthogonal/unitary matrix, Z is a p-by-p orthogonal/unitary matrix, and R and T assume one of the forms:

$$R = m\begin{pmatrix} \overset{n-m}{0} & \overset{m}{R_{12}} \end{pmatrix}, \quad \text{if } m \le n,$$

or

$$R = \begin{matrix} m-n \\ n \end{matrix} \overset{n}{\begin{pmatrix} R_{11} \\ R_{21} \end{pmatrix}}, \quad \text{if } m > n$$

where R_{11} or R_{21} is upper triangular, and

$$T = \begin{matrix} n \\ p-n \end{matrix} \overset{n}{\begin{pmatrix} T_{11} \\ 0 \end{pmatrix}}, \quad \text{if } p \ge n$$

or

$$T = \begin{array}{c} \\ p \end{array} \begin{array}{cc} p & n-p \\ (T_{11} & T_{12}) \end{array} \quad , \text{ if } p < n \ ,$$

where T_{11} is upper triangular.

In particular, if *B* is square and nonsingular, the *GRQ* factorization of *A* and *B* implicitly gives the *RQ* factorization of AB^{-1} as:

$$AB^{-1} = (R\ T^{-1})\ Z^H$$

Input Parameters

m	`INTEGER`. The number of rows of the matrix *A* ($m \geq 0$).
p	`INTEGER`. The number of rows in *B* ($p \geq 0$).
n	`INTEGER`. The number of columns of the matrices *A* and *B* ($n \geq 0$).
a, b, work	`REAL` for `sggrqf` `DOUBLE PRECISION` for `dggrqf` `COMPLEX` for `cggrqf` `DOUBLE COMPLEX` for `zggrqf`. Arrays: `a(lda,*)` contains the *m*-by-*n* matrix *A*. The second dimension of `a` must be at least max(1, *n*). `b(ldb,*)` contains the *p*-by-*n* matrix *B*. The second dimension of `b` must be at least max(1, *n*). `work(lwork)` is a workspace array.
lda	`INTEGER`. The first dimension of `a`; at least max(1, *m*).
ldb	`INTEGER`. The first dimension of `b`; at least max(1, *p*).
lwork	`INTEGER`. The size of the `work` array; must be at least max(1, *n*, *m*, *p*) See *Application notes* for the suggested value of `lwork`.

Output Parameters

a, b	Overwritten by the factorization data as follows:

on exit, if $m \le n$, the upper triangle of the subarray `a`(1:`m`, `n`-`m`+1:`n`) contains the `m`-by-`m` upper triangular matrix *R*;
if $m > n$, the elements on and above the (`m`-`n`)th subdiagonal contain the `m`-by-`n` upper trapezoidal matrix *R;* the remaining elements, with the array `taua`, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors;
the elements on and above the diagonal of the array `b` contain the min(`p`,`n`)-by-`n` upper trapezoidal matrix *T* (*T* is upper triangular if $p \ge n$); the elements below the diagonal, with the array `taub`, represent the orthogonal/unitary matrix *Z* as a product of elementary reflectors.

`taua, taub` — `REAL` for `sggrqf`
`DOUBLE PRECISION` for `dggrqf`
`COMPLEX` for `cggrqf`
`DOUBLE COMPLEX` for `zggrqf`.
Arrays, `DIMENSION` at least max (1, min(`m`, `n`)) for `taua` and at least max (1, min(`p`, `n`)) for `taub`.
The array `taua` contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q*.

The array `taub` contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Z*.

`work(1)` — If `info` = 0, on exit `work(1)` contains the minimum value of `lwork` required for optimum performance. Use this `lwork` for subsequent runs.

`info` — `INTEGER`.
If `info` = 0, the execution is successful.
If `info` = `-i`, the `i`th parameter had an illegal value.

Application Notes

For better performance, try using
`lwork` ≥ max(`n`,`m`,`p`)*max(*nb1,nb2,nb3*),

where *nb1* is the optimal blocksize for the *RQ* factorization of an `m`-by-`n` matrix, *nb2* is the optimal blocksize for the *QR* factorization of an `p`-by-`n` matrix, and *nb3* is the optimal blocksize for a call of `?ormrq/?unmrq`.
If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

Singular Value Decomposition

This section describes LAPACK routines for computing the *singular value decomposition* (SVD) of a general m by n matrix A:

$$A = U\Sigma V^H.$$

In this decomposition, U and V are unitary (for complex A) or orthogonal (for real A); Σ is an m by n diagonal matrix with real diagonal elements σ_i:

$$\sigma_1 \ge \sigma_2 \ge \ldots \ge \sigma_{\min(m,\,n)} \ge 0.$$

The diagonal elements σ_i are *singular values* of A. The first min(m, n) columns of the matrices U and V are, respectively, *left* and *right singular vectors* of A. The singular values and singular vectors satisfy

$$Av_i = \sigma_i u_i \text{ and } A^H u_i = \sigma_i v_i$$

where u_i and v_i are the ith columns of U and V, respectively.

To find the SVD of a general matrix A, call the LAPACK routine `?gebrd` or `?gbbrd` for reducing A to a bidiagonal matrix B by a unitary (orthogonal) transformation: $A = QBP^H$. Then call `?bdsqr`, which forms the SVD of a bidiagonal matrix: $B = U_1\Sigma V_1^H$.

Thus, the sought-for SVD of A is given by $A = U\Sigma V^H = (QU_1)\,\Sigma\,(V_1^H P^H)$.

Table 4-2 Computational Routines for Singular Value Decomposition (SVD)

Operation	Real matrices	Complex matrices
Reduce A to a bidiagonal matrix B: $A = QBP^H$ (full storage)	`?gebrd`	`?gebrd`
Reduce A to a bidiagonal matrix B: $A = QBP^H$ (band storage)	`?gbbrd`	`?gbbrd`
Generate the orthogonal (unitary) matrix Q or P	`?orgbr`	`?ungbr`
Apply the orthogonal (unitary) matrix Q or P	`?ormbr`	`?unmbr`
Form singular value decomposition of the bidiagonal matrix B: $B = U\Sigma V^H$	`?bdsqr` `?bdsdc`	`?bdsqr`

Figure 4-1 Decision Tree: Singular Value Decomposition

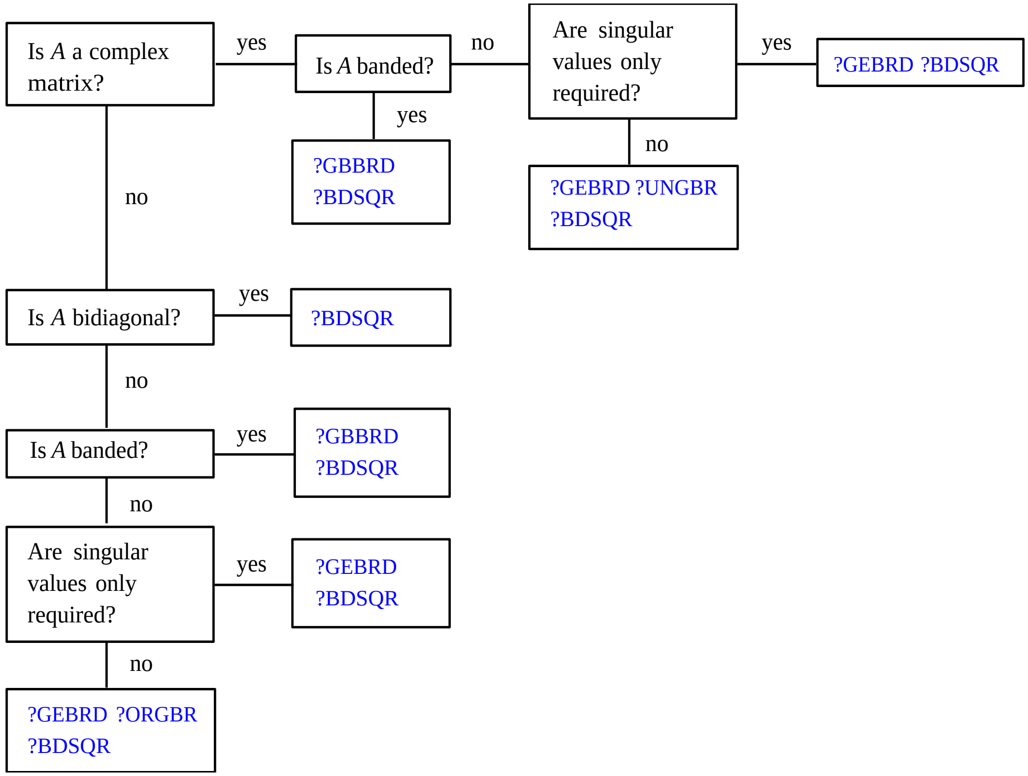


Figure 4-1 presents a decision tree that helps you choose the right sequence of routines for SVD, depending on whether you need singular values only or singular vectors as well, whether *A* is real or complex, and so on.

You can use the SVD to find a minimum-norm solution to a (possibly) rank-deficient least-squares problem of minimizing $||Ax - b||_2$. The effective rank *k* of the matrix *A* can be determined as the number of singular values which exceed a suitable threshold. The minimum-norm solution is

$$x = V_k(\Sigma_k)^{-1}c$$

where Σ_k is the leading *k* by *k* submatrix of Σ, the matrix V_k consists of the first *k* columns of $V = PV_1$, and the vector *c* consists of the first *k* elements of $U^Hb = U_1^HQ^Hb$.

?gebrd

Reduces a general matrix to bidiagonal form.

Syntax

```
call sgebrd ( m, n, a, lda, d, e, tauq, taup, work, lwork, info )
call dgebrd ( m, n, a, lda, d, e, tauq, taup, work, lwork, info )
call cgebrd ( m, n, a, lda, d, e, tauq, taup, work, lwork, info )
call zgebrd ( m, n, a, lda, d, e, tauq, taup, work, lwork, info )
```

Description

The routine reduces a general m by n matrix A to a bidiagonal matrix B by an orthogonal (unitary) transformation.

If $m \ge n$, the reduction is given by $A = QBP^H = Q\begin{pmatrix} B_1 \\ 0 \end{pmatrix}P^H = Q_1 B_1 P^H$,

where B_1 is an n by n upper diagonal matrix, Q and P are orthogonal or, for a complex A, unitary matrices; Q_1 consists of the first n columns of Q.

If $m < n$, the reduction is given by

$$A = QBP^H = Q(B_1 0)P^H = Q_1 B_1 P_1^H,$$

where B_1 is an m by m lower diagonal matrix, Q and P are orthogonal or, for a complex A, unitary matrices; P_1 consists of the first m rows of P.

The routine does not form the matrices Q and P explicitly, but represents them as products of elementary reflectors. Routines are provided to work with the matrices Q and P in this representation:

If the matrix A is real,

- to compute Q and P explicitly, call `?orgbr`.
- to multiply a general matrix by Q or P, call `?ormbr`.

If the matrix A is complex,

- to compute Q and P explicitly, call `?ungbr`.
- to multiply a general matrix by Q or P, call `?unmbr`.

Input Parameters

m	`INTEGER`. The number of rows in the matrix *A* ($m \ge 0$).
n	`INTEGER`. The number of columns in *A* ($n \ge 0$).
a, work	`REAL` for `sgebrd` `DOUBLE PRECISION` for `dgebrd` `COMPLEX` for `cgebrd` `DOUBLE COMPLEX` for `zgebrd`. Arrays: `a(lda,*)` contains the matrix *A*. The second dimension of `a` must be at least max(1, *n*). `work(lwork)` is a workspace array.
lda	`INTEGER`. The first dimension of `a`; at least max(1, *m*).
lwork	`INTEGER`. The dimension of `work`; at least max(1, *m*, *n*). See *Application notes* for the suggested value of `lwork`.

Output Parameters

a	If $m \ge n$, the diagonal and first super-diagonal of `a` are overwritten by the upper bidiagonal matrix *B*. Elements below the diagonal are overwritten by details of *Q*, and the remaining elements are overwritten by details of *P*. If $m < n$, the diagonal and first sub-diagonal of `a` are overwritten by the lower bidiagonal matrix *B*. Elements above the diagonal are overwritten by details of *P*, and the remaining elements are overwritten by details of *Q*.
d	`REAL` for single-precision flavors `DOUBLE PRECISION` for double-precision flavors. Array, `DIMENSION` at least max(1, min(*m*, *n*)). Contains the diagonal elements of *B*.
e	`REAL` for single-precision flavors `DOUBLE PRECISION` for double-precision flavors. Array, `DIMENSION` at least max(1, min(*m*, *n*) – 1). Contains the off-diagonal elements of *B*.
tauq, taup	`REAL` for `sgebrd` `DOUBLE PRECISION` for `dgebrd` `COMPLEX` for `cgebrd`

	`DOUBLE COMPLEX` for `zgebrd`. Arrays, `DIMENSION` at least max (1, min(m, n)). Contain further details of the matrices Q and P.
`work``(1)`	If *`info`* = 0, on exit *`work`*`(1)` contains the minimum value of *`lwork`* required for optimum performance. Use this *`lwork`* for subsequent runs.
`info`	`INTEGER`. If *`info`* = 0, the execution is successful. If *`info`* = `-`*`i`*, the *`i`*th parameter had an illegal value.

Application Notes

For better performance, try using *`lwork`* $= (m + n)*$*blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *`lwork`* for the first run. On exit, examine *`work`*`(1)` and use this value for subsequent runs.

The computed matrices Q, B, and P satisfy $QBP^H = A + E$, where $||E||_2 = c(n)\varepsilon\ ||A||_2$, $c(n)$ is a modestly increasing function of n, and ε is the machine precision.

The approximate number of floating-point operations for real flavors is
$(4/3)*n^2*(3*m - n)$ for $m \geq n$,
$(4/3)*m^2*(3*n - m)$ for $m < n$.
The number of operations for complex flavors is four times greater.

If n is much less than m, it can be more efficient to first form the QR factorization of A by calling `?geqrf` and then reduce the factor R to bidiagonal form. This requires approximately $2*n^2*(m + n)$ floating-point operations.

If m is much less than n, it can be more efficient to first form the LQ factorization of A by calling `?gelqf` and then reduce the factor L to bidiagonal form. This requires approximately $2*m^2*(m + n)$ floating-point operations.

?gbbrd

Reduces a general band matrix to bidiagonal form.

Syntax

```
call sgbbrd ( vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt,
    ldpt, c, ldc, work, info )
call dgbbrd ( vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt,
    ldpt, c, ldc, work, info )
call cgbbrd ( vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt,
    ldpt, c, ldc, work, rwork, info )
call zgbbrd ( vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt,
    ldpt, c, ldc, work, rwork, info )
```

Description

This routine reduces an *m* by *n* band matrix A to upper bidiagonal matrix B: $A = QBP^H$. Here the matrices Q and P are orthogonal (for real A) or unitary (for complex A). They are determined as products of Givens rotation matrices, and may be formed explicitly by the routine if required. The routine can also update a matrix C as follows: $C = Q^H C$.

Input Parameters

vect — `CHARACTER*1`. Must be `'N'` or `'Q'` or `'P'` or `'B'`.
If *vect* = `'N'`, neither Q nor P^H is generated.
If *vect* = `'Q'`, the routine generates the matrix Q.
If *vect* = `'P'`, the routine generates the matrix P^H.
If *vect* = `'B'`, the routine generates both Q and P^H.

m — `INTEGER`. The number of rows in the matrix A ($m \geq 0$).

n — `INTEGER`. The number of columns in A ($n \geq 0$).

ncc — `INTEGER`. The number of columns in C ($ncc \geq 0$).

kl — `INTEGER`. The number of sub-diagonals within the band of A ($kl \geq 0$).

ku — `INTEGER`. The number of super-diagonals within the band of A ($ku \geq 0$).

ab,c,work — `REAL` for `sgbbrd`
`DOUBLE PRECISION` for `dgbbrd`
`COMPLEX` for `cgbbrd`
`DOUBLE COMPLEX` for `zgbbrd`.

Arrays:
ab(*ldab*,*) contains the matrix *A* in band storage (see Matrix Storage Schemes).
The second dimension of *a* must be at least max(1, *n*).

c(*ldc*,*) contains an *m* by *ncc* matrix *C*.
If *ncc* = 0, the array *c* is not referenced. The second dimension of *c* must be at least max(1, *ncc*).

work(*) is a workspace array.
The dimension of *work* must be at least 2*max(*m*, *n*) for real flavors, or max(*m*, *n*) for complex flavors.

ldab — INTEGER. The first dimension of the array *ab* ($ldab \ge kl + ku + 1$).

ldq — INTEGER. The first dimension of the output array *q*.
$ldq \ge \max(1, m)$ if *vect* = 'Q' or 'B',
$ldq \ge 1$ otherwise.

ldpt — INTEGER. The first dimension of the output array *pt*.
$ldpt \ge \max(1, n)$ if *vect* = 'P' or 'B',
$ldpt \ge 1$ otherwise.

ldc — INTEGER. The first dimension of the array *c*.
$ldc \ge \max(1, m)$ if $ncc > 0$; $ldc \ge 1$ if $ncc = 0$.

rwork — REAL for cgbbrd
DOUBLE PRECISION for zgbbrd.
A workspace array, DIMENSION at least max(*m*, *n*).

Output Parameters

ab — Overwritten by values generated during the reduction.

d — REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least max(1, min(*m*, *n*)).
Contains the diagonal elements of the matrix *B*.

e — REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least max(1, min(*m*, *n*) – 1).
Contains the off-diagonal elements of *B*.

q, pt REAL for sgebrd
DOUBLE PRECISION for dgebrd
COMPLEX for cgebrd
DOUBLE COMPLEX for zgebrd.
Arrays:

q(ldq,*) contains the output *m* by *m* matrix *Q*.
The second dimension of *q* must be at least max(1, *m*).

p(ldpt,*) contains the output *n* by *n* matrix P^H.
The second dimension of *pt* must be at least max(1, *n*).

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The computed matrices *Q*, *B,* and *P* satisfy $QBP^H = A + E$, where $||E||_2 = c(n)\varepsilon\ ||A||_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision.

If $m = n$, the total number of floating-point operations for real flavors is approximately the sum of:

$6*n^2*(kl + ku)$ if *vect* = 'N' and *ncc* = 0,

$3*n^2*ncc*(kl + ku - 1)/(kl + ku)$ if *C* is updated, and

$3*n^3*(kl + ku - 1)/(kl + ku)$ if either *Q* or P^H is generated (double this if both).

To estimate the number of operations for complex flavors, use the same formulas with the coefficients 20 and 10 (instead of 6 and 3).

?orgbr

Generates the real orthogonal matrix Q or P^T determined by `?gebrd`.

Syntax

```
call sorgbr ( vect, m, n, k, a, lda, tau, work, lwork, info )
call dorgbr ( vect, m, n, k, a, lda, tau, work, lwork, info )
```

Description

The routine generates the whole or part of the orthogonal matrices Q and P^T formed by the routines `sgebrd`/`dgebrd` (see page 4-70). Use this routine after a call to `sgebrd`/`dgebrd`. All valid combinations of arguments are described in *Input parameters*. In most cases you'll need the following:

To compute the whole *m* by *m* matrix Q:
`call ?orgbr ( 'Q', m, m, n, a ... )`
(note that the array *a* must have at least *m* columns).

To form the *n* leading columns of Q if $m > n$:
`call ?orgbr ( 'Q', m, n, n, a ... )`

To compute the whole *n* by *n* matrix P^T:
`call ?orgbr ( 'P', n, n, m, a ... )`
(note that the array *a* must have at least *n* rows).

To form the *m* leading rows of P^T if $m < n$:
`call ?orgbr ( 'P', m, n, m, a ... )`

Input Parameters

vect — `CHARACTER*1`. Must be `'Q'` or `'P'`.
If *vect* = `'Q'`, the routine generates the matrix Q.
If *vect* = `'P'`, the routine generates the matrix P^T.

m — `INTEGER`. The number of required rows of Q or P^T.

n — `INTEGER`. The number of required columns of Q or P^T.

k — INTEGER. One of the dimensions of *A* in ?gebrd:
If *vect* = 'Q', the number of columns in *A*;
If *vect* = 'P', the number of rows in *A*.

Constraints: $m \geq 0$, $n \geq 0$, $k \geq 0$.
For *vect* ='Q': $k \leq n \leq m$ if $m > k$, or $m = n$ if $m \leq k$.
For *vect* ='P': $k \leq m \leq n$ if $n > k$, or $m = n$ if $n \leq k$.

a, work — REAL for sorgbr
DOUBLE PRECISION for dorgbr.
Arrays:
a(*lda*,*) is the array *a* as returned by ?gebrd.
The second dimension of *a* must be at least max(1, *n*).

work(*lwork*) is a workspace array.

lda — INTEGER. The first dimension of *a*; at least max(1, *m*).

tau — REAL for sorgbr
DOUBLE PRECISION for dorgbr.
For *vect* = 'Q', the array *tauq* as returned by ?gebrd. For *vect* = 'P', the array *taup* as returned by ?gebrd.
The dimension of *tau* must be at least max(1, min(*m*,*k*)) for *vect* ='Q', or max(1, min(*m*, *k*)) for *vect* = 'P'.

lwork — INTEGER. The size of the *work* array.
See *Application notes* for the suggested value of *lwork*.

Output Parameters

a — Overwritten by the orthogonal matrix *Q* or P^T (or the leading rows or columns thereof) as specified by *vect*, *m*, and *n*.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* = min(*m*, *n*)**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $||E||_2 = O(\varepsilon)$.

The approximate numbers of floating-point operations for the cases listed in *Description* are as follows:

To form the whole of Q:

$(4/3)n(3m^2 - 3m*n + n^2)$	if $m > n$;
$(4/3)m^3$	if $m \le n$.

To form the n leading columns of Q when $m > n$:

$(2/3)n^2(3m - n^2)$	if $m > n$.

To form the whole of P^T:

$(4/3)n^3$	if $m \ge n$;
$(4/3)m(3n^2 - 3m*n + m^2)$	if $m < n$.

To form the m leading columns of P^T when $m < n$:

$(2/3)n^2(3m - n^2)$	if $m > n$.

The complex counterpart of this routine is ?ungbr.

?ormbr

Multiplies an arbitrary real matrix by the real orthogonal matrix Q or P^T determined by `?gebrd`.

Syntax

```
call sormbr (vect,side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info)
call dormbr (vect,side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info)
```

Description

Given an arbitrary real matrix C, this routine forms one of the matrix products QC, Q^TC, CQ, CQ^T, PC, P^TC, CP, or CP^T, where Q and P are orthogonal matrices computed by a call to `sgebrd`/`dgebrd` (see page 4-70). The routine overwrites the product on C.

Input Parameters

In the descriptions below, *r* denotes the order of Q or P^T:
If *side* = 'L', *r* = *m*; if *side* = 'R', *r* = *n*.

vect — `CHARACTER*1`. Must be 'Q' or 'P'.
If *vect* = 'Q', then Q or Q^T is applied to C.
If *vect* = 'P', then P or P^T is applied to C.

side — `CHARACTER*1`. Must be 'L' or 'R'.
If *side* = 'L', multipliers are applied to C from the left.
If *side* = 'R', they are applied to C from the right.

trans — `CHARACTER*1`. Must be 'N' or 'T'.
If *trans* = 'N', then Q or P is applied to C.
If *trans* = 'T', then Q^T or P^T is applied to C.

m — `INTEGER`. The number of rows in C.

n — `INTEGER`. The number of columns in C.

k — `INTEGER`. One of the dimensions of A in `?gebrd`:
If *vect* = 'Q', the number of columns in A;
If *vect* = 'P', the number of rows in A.

Constraints: $m \ge 0$, $n \ge 0$, $k \ge 0$.

a, c, work	`REAL` for `sormbr` `DOUBLE PRECISION` for `dormbr`. Arrays: `a(lda,*)` is the array *a* as returned by `?gebrd`. Its second dimension must be at least max(1, min(r,k)) for *vect* = `'Q'`, or max(1, r)) for *vect* = `'P'`. `c(ldc,*)` holds the matrix C. Its second dimension must be at least max(1, n). `work(lwork)` is a workspace array.
lda	`INTEGER`. The first dimension of *a*. Constraints: *lda* ≥ max(1, r) if *vect* = `'Q'`; *lda* ≥ max(1, min(r,k)) if *vect* = `'P'`.
ldc	`INTEGER`. The first dimension of *c*; *ldc* ≥ max(1, m).
tau	`REAL` for `sormbr` `DOUBLE PRECISION` for `dormbr`. Array, `DIMENSION` at least max (1, min(r, k)). For *vect* = `'Q'`, the array *tauq* as returned by `?gebrd`. For *vect* = `'P'`, the array *taup* as returned by `?gebrd`.
lwork	`INTEGER`. The size of the *work* array. Constraints: *lwork* ≥ max(1, n) if *side* =`'L'`; *lwork* ≥ max(1, m) if *side* =`'R'`. See *Application notes* for the suggested value of *lwork*.

Output Parameters

c	Overwritten by the product QC, Q^TC, CQ, CQ^T, PC, P^TC, CP, or CP^T, as specified by *vect*, *side*, and *trans*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using

`lwork` = `n`**blocksize* for `side` = 'L', or

`lwork` = `m`**blocksize* for `side` = 'R',

where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The computed product differs from the exact product by a matrix E such that $||E||_2 = O(\varepsilon)\ ||C||_2$.

The total number of floating-point operations is approximately

$2*n*k(2*m - k)$	if `side` = 'L' and $m \ge k$;
$2*m*k(2*n - k)$	if `side` = 'R' and $n \ge k$;
$2*m^2*n$	if `side` = 'L' and $m < k$;
$2*n^2*m$	if `side` = 'R' and $n < k$.

The complex counterpart of this routine is `?unmbr`.

?ungbr

Generates the complex unitary matrix Q or P^H determined by `?gebrd`.

Syntax

```
call cungbr ( vect, m, n, k, a, lda, tau, work, lwork, info )
call zungbr ( vect, m, n, k, a, lda, tau, work, lwork, info )
```

Description

The routine generates the whole or part of the unitary matrices Q and P^H formed by the routines `cgebrd`/`zgebrd` (see page 4-70). Use this routine after a call to `cgebrd`/`zgebrd`. All valid combinations of arguments are described in *Input Parameters*; in most cases you'll need the following:

To compute the whole *m* by *m* matrix Q:
`call ?ungbr ( 'Q', m, m, n, a ... )`
(note that the array *a* must have at least *m* columns).

To form the *n* leading columns of Q if $m > n$:
`call ?ungbr ( 'Q', m, n, n, a ... )`

To compute the whole *n* by *n* matrix P^H:
`call ?ungbr ( 'P', n, n, m, a ... )`
(note that the array *a* must have at least *n* rows).

To form the *m* leading rows of P^H if $m < n$:
`call ?ungbr ( 'P', m, n, m, a ... )`

Input Parameters

vect	`CHARACTER*1`. Must be `'Q'` or `'P'`. If *vect* = `'Q'`, the routine generates the matrix Q. If *vect* = `'P'`, the routine generates the matrix P^H.
m	`INTEGER`. The number of required rows of Q or P^H.
n	`INTEGER`. The number of required columns of Q or P^H.

k	`INTEGER`. One of the dimensions of *A* in `?gebrd`: If *vect* = `'Q'`, the number of columns in *A*; If *vect* = `'P'`, the number of rows in *A*. Constraints: $m \ge 0$, $n \ge 0$, $k \ge 0$. For *vect* =`'Q'`: $k \le n \le m$ if $m > k$, or $m = n$ if $m \le k$. For *vect* =`'P'`: $k \le m \le n$ if $n > k$, or $m = n$ if $n \le k$.
a, work	`COMPLEX` for `cungbr` `DOUBLE COMPLEX` for `zungbr`. Arrays: `a(lda,*)` is the array `a` as returned by `?gebrd`. The second dimension of `a` must be at least max(1, *n*). `work(lwork)` is a workspace array.
lda	`INTEGER`. The first dimension of `a`; at least max(1, *m*).
tau	`COMPLEX` for `cungbr` `DOUBLE COMPLEX` for `zungbr`. For *vect* = `'Q'`, the array *tauq* as returned by `?gebrd`. For *vect* = `'P'`, the array *taup* as returned by `?gebrd`. The dimension of *tau* must be at least max(1, min(*m*,*k*)) for *vect* =`'Q'`, or max(1, min(*m*, *k*)) for *vect* = `'P'`.
lwork	`INTEGER`. The size of the *work* array. Constraint: *lwork* ≥ max(1, min(*m*, *n*)). See *Application notes* for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the orthogonal matrix *Q* or P^T (or the leading rows or columns thereof) as specified by *vect*, *m*, and *n*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* = min(*m, n*)**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $||E||_2 = O(\varepsilon)$.

The approximate numbers of floating-point operations for the cases listed in *Description* are as follows:

To form the whole of Q:

$(16/3)n(3m^2 - 3m*n + n^2)$ if $m > n$;

$(16/3)m^3$ if $m \le n$.

To form the n leading columns of Q when $m > n$:

$(8/3)n^2(3m - n^2)$ if $m > n$.

To form the whole of P^T:

$(16/3)n^3$ if $m \ge n$;

$(16/3)m(3n^2 - 3m*n + m^2)$ if $m < n$.

To form the m leading columns of P^T when $m < n$:

$(8/3)n^2(3m - n^2)$ if $m > n$.

The real counterpart of this routine is ?orgbr.

?unmbr

Multiplies an arbitrary complex matrix by the unitary matrix Q or P determined by `?gebrd`.

Syntax

```
call cunmbr (vect,side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info)
call zunmbr (vect,side,trans,m,n,k,a,lda,tau,c,ldc,work,lwork,info)
```

Description

Given an arbitrary complex matrix C, this routine forms one of the matrix products QC, $Q^H C$, CQ, CQ^H, PC, $P^H C$, CP, or CP^H, where Q and P are orthogonal matrices computed by a call to `cgebrd`/`zgebrd` (see page 4-70). The routine overwrites the product on C.

Input Parameters

In the descriptions below, *r* denotes the order of Q or P^H:
If *side* = 'L', *r* = *m*; if *side* = 'R', *r* = *n*.

vect — `CHARACTER*1`. Must be 'Q' or 'P'.
If *vect* = 'Q', then Q or Q^H is applied to C.
If *vect* = 'P', then P or P^H is applied to C.

side — `CHARACTER*1`. Must be 'L' or 'R'.
If *side* = 'L', multipliers are applied to C from the left.
If *side* = 'R', they are applied to C from the right.

trans — `CHARACTER*1`. Must be 'N' or 'C'.
If *trans* = 'N', then Q or P is applied to C.
If *trans* = 'C', then Q^H or P^H is applied to C.

m — `INTEGER`. The number of rows in C.

n — `INTEGER`. The number of columns in C.

k — `INTEGER`. One of the dimensions of A in `?gebrd`:
If *vect* = 'Q', the number of columns in A;
If *vect* = 'P', the number of rows in A.

Constraints: $m \ge 0$, $n \ge 0$, $k \ge 0$.

a, c, work — `COMPLEX` for `cunmbr`
`DOUBLE COMPLEX` for `zunmbr`.
Arrays:
a(*lda*,*) is the array *a* as returned by `?gebrd`.
Its second dimension must be at least max(1, min(r,k)) for *vect* = `'Q'`, or max(1, r)) for *vect* = `'P'`.

c(*ldc*,*) holds the matrix *C*.
Its second dimension must be at least max(1, n).

work(*lwork*) is a workspace array.

lda — `INTEGER`. The first dimension of *a*. Constraints:
lda ≥ max(1, r) if *vect* = `'Q'`;
lda ≥ max(1, min(r,k)) if *vect* = `'P'`.

ldc — `INTEGER`. The first dimension of *c*; *ldc* ≥ max(1, m).

tau — `COMPLEX` for `cunmbr`
`DOUBLE COMPLEX` for `zunmbr`.
Array, `DIMENSION` at least max (1, min(r, k)).
For *vect* = `'Q'`, the array *tauq* as returned by `?gebrd`. For *vect* = `'P'`, the array *taup* as returned by `?gebrd`.

lwork — `INTEGER`. The size of the *work* array. Constraints:
lwork ≥ max(1, n) if *side* = `'L'`;
lwork ≥ max(1, m) if *side* = `'R'`.
See *Application notes* for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product QC, Q^HC, CQ, CQ^H, PC, P^HC, CP, or CP^H, as specified by *vect*, *side*, and *trans*.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using

`lwork` = `n`*_blocksize_ for `side` = 'L', or

`lwork` = `m`*_blocksize_ for `side` = 'R',

where _blocksize_ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the _blocked algorithm_.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The computed product differs from the exact product by a matrix E such that $||E||_2 = O(\varepsilon) ||C||_2$.

The total number of floating-point operations is approximately

$8*n*k(2*m - k)$	if `side` = 'L' and $m \geq k$;
$8*m*k(2*n - k)$	if `side` = 'R' and $n \geq k$;
$8*m^2*n$	if `side` = 'L' and $m < k$;
$8*n^2*m$	if `side` = 'R' and $n < k$.

The real counterpart of this routine is `?ormbr`.

?bdsqr

Computes the singular value decomposition of a general matrix that has been reduced to bidiagonal form.

Syntax

```
call sbdsqr ( uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu,
   c, ldc, work, info )
call dbdsqr ( uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu,
   c, ldc, work, info )
call cbdsqr ( uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu,
   c, ldc, work, info )
call zbdsqr ( uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu,
   c, ldc, work, info )
```

Description

This routine computes the singular values and, optionally, the right and/or left singular vectors from the Singular Value Decomposition (SVD) of a real *n*-by-*n* (upper or lower) bidiagonal matrix *B* using the implicit zero-shift *QR* algorithm. The SVD of *B* has the form $B = Q * S * P^H$ where *S* is the diagonal matrix of singular values, *Q* is an orthogonal matrix of left singular vectors, and *P* is an orthogonal matrix of right singular vectors. If left singular vectors are requested, this subroutine actually returns $U * Q$ instead of *Q*, and, if right singular vectors are requested, this subroutine returns
$P^H * VT$ instead of P^H, for given real/complex input matrices *U* and *VT*. When *U* and *VT* are the orthogonal/unitary matrices that reduce a general matrix *A* to bidiagonal form: $A = U * B * VT$, as computed by `?gebrd`, then

$$A = (U * Q) * S * (P^H * VT)$$

is the SVD of *A*. Optionally, the subroutine may also compute $Q^H * C$ for a given real/complex input matrix *C*.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, *B* is an upper bidiagonal matrix.
If *uplo* = `'L'`, *B* is a lower bidiagonal matrix.

n — `INTEGER`. The order of the matrix *B* ($n \geq 0$).

ncvt	`INTEGER`. The number of columns of the matrix *VT*, that is, the number of right singular vectors (*ncvt* ≥ 0). Set *ncvt* = 0 if no right singular vectors are required.
nru	`INTEGER`. The number of rows in *U*, that is, the number of left singular vectors (*nru* ≥ 0). Set *nru* = 0 if no left singular vectors are required.
ncc	`INTEGER`. The number of columns in the matrix *C* used for computing the product $Q^H C$ (*ncc* ≥ 0). Set *ncc* = 0 if no matrix *C* is supplied.
d, e, work	`REAL` for single-precision flavors `DOUBLE PRECISION` for double-precision flavors. Arrays: *d*(*) contains the diagonal elements of *B*. The dimension of *d* must be at least max(1, *n*). *e*(*) contains the (*n*-1) off-diagonal elements of *B*. The dimension of *e* must be at least max(1, *n*). *e*(*n*) is used for workspace. *work*(*) is a workspace array. The dimension of *work* must be at least max(1, 2*n) if *ncvt* = *nru* = *ncc* = 0; max(1, 4*(*n*-1)) otherwise.
vt, u, c	`REAL` for `sbdsqr` `DOUBLE PRECISION` for `dbdsqr` `COMPLEX` for `cbdsqr` `DOUBLE COMPLEX` for `zbdsqr`. Arrays: *vt*(*ldvt*,*) contains an *n* by *ncvt* matrix *VT*. The second dimension of *vt* must be at least max(1, *ncvt*). *vt* is not referenced if *ncvt* = 0. *u*(*ldu*,*) contains an *nru* by *n* unit matrix *U*. The second dimension of *u* must be at least max(1, *n*). *u* is not referenced if *nru* = 0. *c*(*ldc*,*) contains the matrix *C* for computing the product $Q^H * C$. The second dimension of *c* must be at least max(1,*ncc*). The array is not referenced if *ncc* = 0.

ldvt INTEGER. The first dimension of *vt*. Constraints:
ldvt ≥ max(1, *n*) if *ncvt* > 0;
ldvt ≥ 1 if *ncvt* = 0.

ldu INTEGER. The first dimension of *u*. Constraint:
ldu ≥ max(1, *nru*).

ldc INTEGER. The first dimension of *c*. Constraints:
ldc ≥ max(1, *n*) if *ncc* > 0;
ldc ≥ 1 otherwise.

Output Parameters

d On exit, if *info* = 0, overwritten by the singular values in decreasing order (see *info*).

e On exit, if *info* = 0, *e* is destroyed. See also *info* below.

c Overwritten by the product $Q^H * C$.

vt On exit, this array is overwritten by $P^H * VT$.

u On exit, this array is overwritten by $U * Q$.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.
If *info* = *i*, the algorithm failed to converge;
i specifies how many off-diagonals did not converge.
In this case, *d* and *e* contain on exit the diagonal and off-diagonal elements, respectively, of a bidiagonal matrix orthogonally equivalent to *B*.

Application Notes

Each singular value and singular vector is computed to high relative accuracy. However, the reduction to bidiagonal form (prior to calling the routine) may decrease the relative accuracy in the small singular values of the original matrix if its singular values vary widely in magnitude.

If σ_i is an exact singular value of *B*, and s_i is the corresponding computed value, then

$$|s_i - \sigma_i| \le p(m, n)\varepsilon\sigma_i$$

where $p(m, n)$ is a modestly increasing function of *m* and *n*, and ε is the machine precision. If only singular values are computed, they are computed more accurately than when some singular vectors are also computed (that is, the function $p(m, n)$ is smaller).

If u_i is the corresponding exact left singular vector of B, and w_i is the corresponding computed left singular vector, then the angle $\theta(u_i, w_i)$ between them is bounded as follows:

$\theta(u_i, w_i) \le p(m, n)\varepsilon / \min_{i\ne j}(|\sigma_i - \sigma_j|/|\sigma_i + \sigma_j|)$.

Here $\min_{i\ne j}(|\sigma_i - \sigma_j|/|\sigma_i + \sigma_j|)$ is the *relative gap* between σ_i and the other singular values. A similar error bound holds for the right singular vectors.

The total number of real floating-point operations is roughly proportional to n^2 if only the singular values are computed. About $6n^2*nru$ additional operations ($12n^2*nru$ for complex flavors) are required to compute the left singular vectors and about $6n^2*ncvt$ operations ($12n^2*ncvt$ for complex flavors) to compute the right singular vectors.

?bdsdc

Computes the singular value decomposition of a real bidiagonal matrix using a divide and conquer method.

Syntax

```
call sbdsdc ( uplo, compq, n, d, e, u, ldu, vt, ldvt, q, iq, work,
              iwork, info )
call dbdsdc ( uplo, compq, n, d, e, u, ldu, vt, ldvt, q, iq, work,
              iwork, info )
```

Description

This routine computes the Singular Value Decomposition (SVD) of a real *n*-by-*n* (upper or lower) bidiagonal matrix *B*: $B = U \Sigma V^T$, using a divide and conquer method, where Σ is a diagonal matrix with non-negative diagonal elements (the singular values of *B*), and *U* and *V* are orthogonal matrices of left and right singular vectors, respectively. `?bdsdc` can be used to compute all singular values, and optionally, singular vectors or singular vectors in compact form.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.

If *uplo* = `'U'`, *B* is an upper bidiagonal matrix.
If *uplo* = `'L'`, *B* is a lower bidiagonal matrix.

compq — `CHARACTER*1`. Must be `'N'`, `'P'`, or `'I'`.

If *compq* = `'N'`, compute singular values only.
If *compq* = `'P'`, compute singular values and compute singular vectors in compact form.
If *compq* = `'I'`, compute singular values and singular vectors.

n — `INTEGER`. The order of the matrix *B* ($n \geq 0$).

d, e, work — `REAL` for `sbdsdc`
`DOUBLE PRECISION` for `sbdsdc`.
Arrays:

d(*) contains the *n* diagonal elements of the bidiagonal matrix *B*. The dimension of *d* must be at least max(1, *n*).

	`e(*)` contains the off-diagonal elements of the bidiagonal matrix *B*. The dimension of `e` must be at least max(1, *n*).
	`work(*)` is a workspace array. The dimension of `work` must be at least: max(1, 4**n*), if `compq` = `'N'`; max(1, 6**n*), if `compq` = `'P'`; max(1, $3*n^2+4*n$), if `compq` = `'I'`.
`ldu`	`INTEGER`. The first dimension of the output array `u`; `ldu` ≥ 1. If singular vectors are desired, then `ldu` ≥ max(1, *n*).
`ldvt`	`INTEGER`. The first dimension of the output array `vt`; `ldvt` ≥ 1. If singular vectors are desired, then `ldvt` ≥ max(1, *n*).
`iwork`	`INTEGER`. Workspace array, dimension at least max(1, 8**n*).

Output Parameters

`d`	If `info` = 0, overwritten by the singular values of *B*.
`e`	On exit, `e` is overwritten.
`u, vt, q`	`REAL` for `sbdsdc` `DOUBLE PRECISION` for `sbdsdc`. Arrays: `u(ldu,*)`, `vt(ldvt,*)`, `q(*)`. If `compq` = `'I'`, then on exit `u` contains the left singular vectors of the bidiagonal matrix *B,* unless `info` ≠ 0 (*see* `info`). For other values of `compq`, `u` is not referenced. The second dimension of `u` must be at least max(1,*n*). If `compq` = `'I'`, then on exit `vt` contains the right singular vectors of the bidiagonal matrix *B,* unless `info` ≠ 0 (*see* `info`). For other values of `compq`, `vt` is not referenced. The second dimension of `vt` must be at least max(1,*n*). If `compq` = `'P'`, then on exit, if `info` = 0, `q` and `iq` contain the left and right singular vectors in a compact form. Specifically, `q` contains all the `REAL` (for `sbdsdc`) or `DOUBLE PRECISION` (for `dbdsdc`) data for singular vectors. For other values of `compq` , `q` is not referenced. See *Application notes* for details.
`iq`	`INTEGER`. Array: `iq(*)`. If `compq` = `'P'`, then on exit, if `info` = 0, `q` and `iq` contain the left and right

singular vectors in a compact form. Specifically, *iq* contains all the INTEGER data for singular vectors. For other values of *compq* , *iq* is not referenced. See *Application notes* for details.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.
If *info* = *i*, the algorithm failed to compute a singular value. The update process of divide and conquer failed.

Symmetric Eigenvalue Problems

Symmetric eigenvalue problems are posed as follows: given an n by n real symmetric or complex Hermitian matrix A, find the *eigenvalues* λ and the corresponding *eigenvectors* z that satisfy the equation

$$Az = \lambda z. \text{ (or, equivalently, } z^H A = \lambda z^H\text{)}.$$

In such eigenvalue problems, all n eigenvalues are real not only for real symmetric but also for complex Hermitian matrices A, and there exists an orthonormal system of n eigenvectors. If A is a symmetric or Hermitian positive-definite matrix, all eigenvalues are positive.

To solve a symmetric eigenvalue problem with LAPACK, you usually need to reduce the matrix to tridiagonal form and then solve the eigenvalue problem with the tridiagonal matrix obtained. LAPACK includes routines for reducing the matrix to a tridiagonal form by an orthogonal (or unitary) similarity transformation $A = QTQ^H$ as well as for solving tridiagonal symmetric eigenvalue problems. These routines are listed in Table 4-3.

There are different routines for symmetric eigenvalue problems, depending on whether you need all eigenvectors or only some of them or eigenvalues only, whether the matrix A is positive-definite or not, and so on.
These routines are based on three primary algorithms for computing eigenvalues and eigenvectors of symmetric problems: the divide and conquer algorithm, the QR algorithm, and bisection followed by inverse iteration. The divide and conquer algorithm is generally more efficient and is recommended for computing all eigenvalues and eigenvectors.
Furthermore, to solve an eigenvalue problem using the divide and conquer algorithm, you need to call only one routine. In general, more than one routine has to be called if the QR algorithm or bisection followed by inverse iteration is used.

Decision tree in Figure 4-2 will help you choose the right routine or sequence of routines for eigenvalue problems with real symmetric matrices. A similar decision tree for complex Hermitian matrices is presented in Figure 4-3.

Figure 4-2 Decision Tree: Real Symmetric Eigenvalue Problems

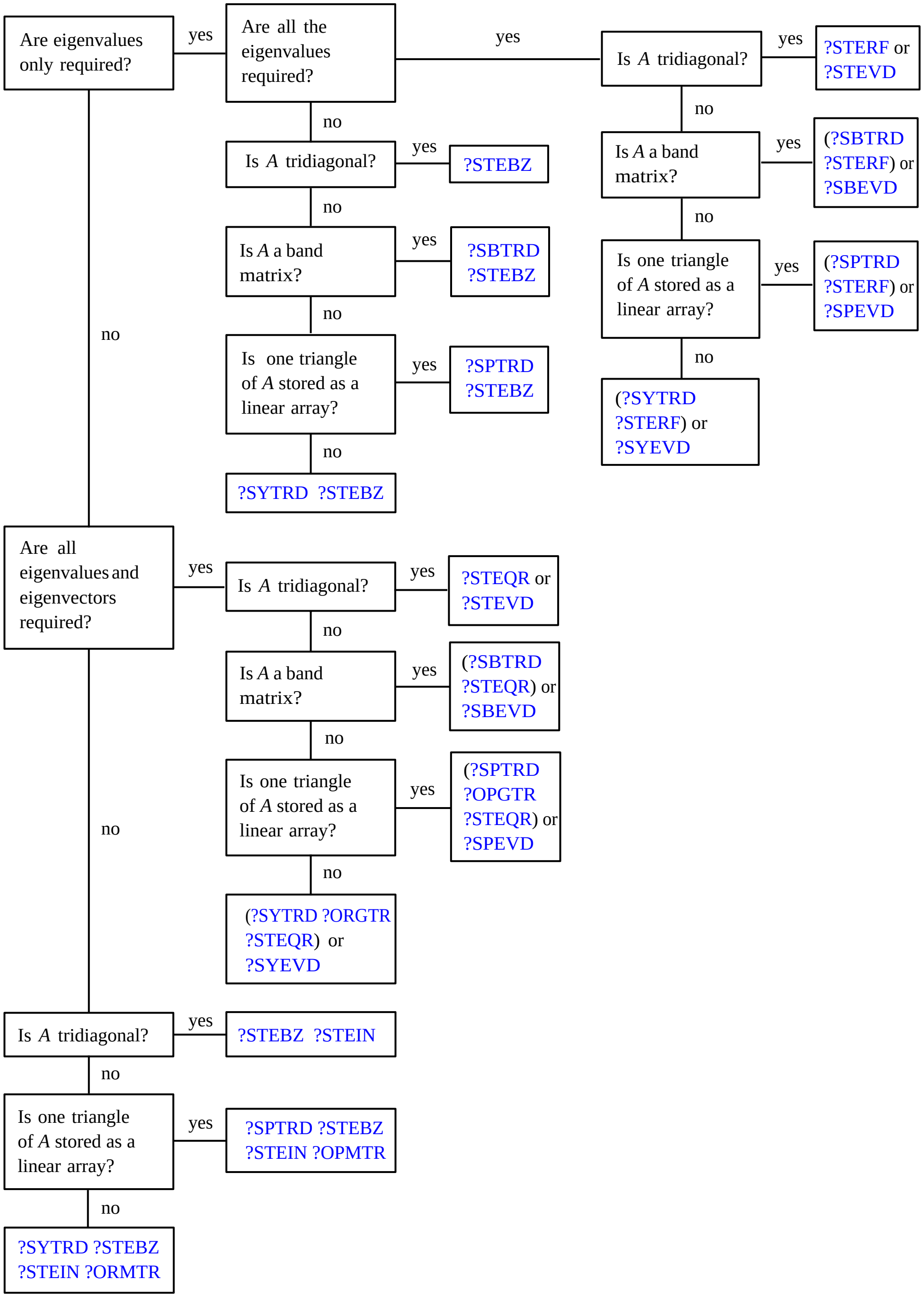


Figure 4-3 Decision Tree: Complex Hermitian Eigenvalue Problems

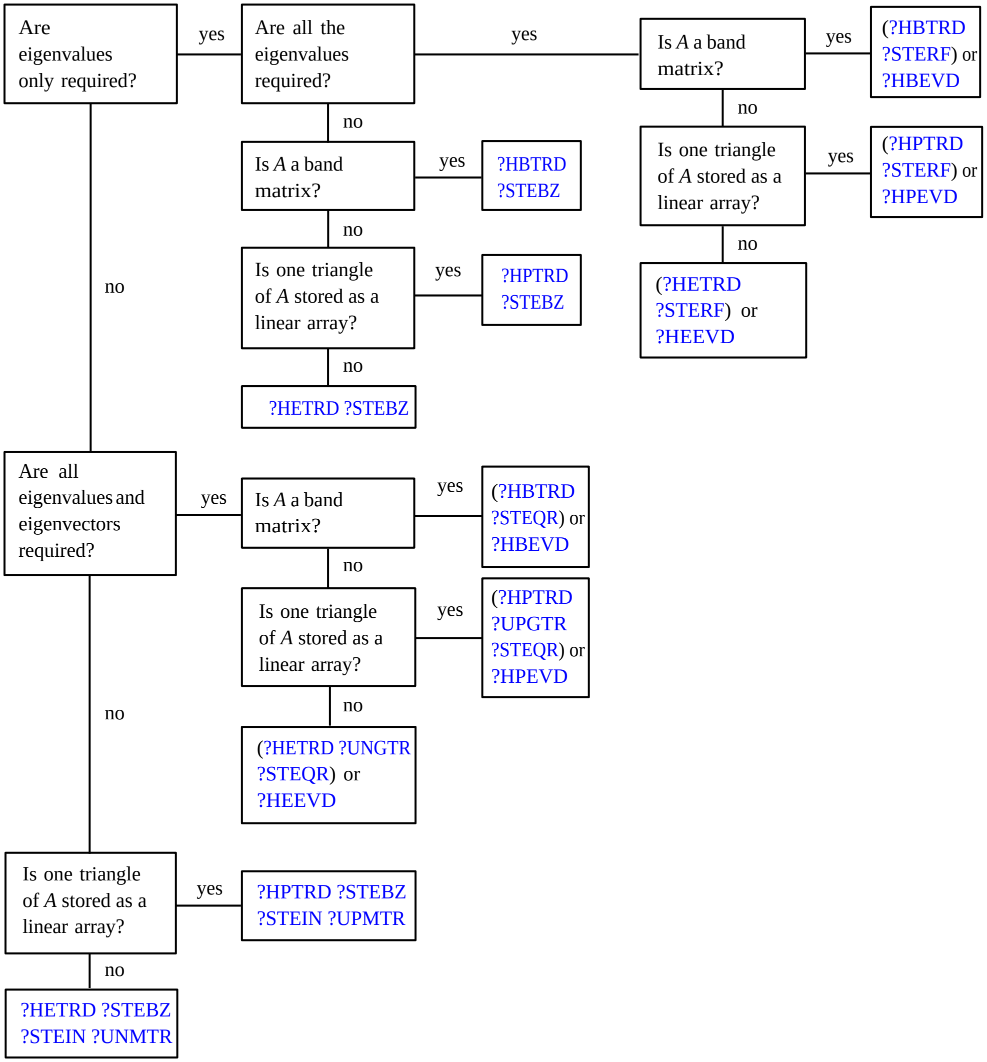


Table 4-3 Computational Routines for Solving Symmetric Eigenvalue Problems

Operation	Real symmetric matrices	Complex Hermitian matrices
Reduce to tridiagonal form $A = QTQ^H$ (full storage)	?sytrd	?hetrd
Reduce to tridiagonal form $A = QTQ^H$ (packed storage)	?sptrd	?hptrd
Reduce to tridiagonal form $A = QTQ^H$ (band storage).	?sbtrd	?hbtrd
Generate matrix *Q* (full storage)	?orgtr	?ungtr
Generate matrix *Q* (packed storage)	?opgtr	?upgtr
Apply matrix *Q* (full storage)	?ormtr	?unmtr
Apply matrix *Q* (packed storage)	?opmtr	?upmtr
Find all eigenvalues of a tridiagonal matrix *T*	?sterf	
Find all eigenvalues and eigenvectors of a tridiagonal matrix *T*	?steqr ?stedc	?steqr ?stedc
Find all eigenvalues and eigenvectors of a tridiagonal positive-definite matrix *T*.	?pteqr	?pteqr
Find selected eigenvalues of a tridiagonal matrix *T*	?stebz ?stegr	?stegr
Find selected eigenvectors of a tridiagonal matrix *T*	?stein ?stegr	?stein ?stegr
Compute the reciprocal condition numbers for the eigenvectors	?disna	?disna

?sytrd

Reduces a real symmetric matrix to tridiagonal form.

Syntax

```
call ssytrd ( uplo,n,a,lda,d,e,tau,work,lwork,info )
call dsytrd ( uplo,n,a,lda,d,e,tau,work,lwork,info )
```

Description

This routine reduces a real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = QTQ^T$. The orthogonal matrix Q is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided for working with Q in this representation. (They are described later in this section.)

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, *a* stores the upper triangular part of A.
If *uplo* = `'L'`, *a* stores the lower triangular part of A.

n — `INTEGER`. The order of the matrix A ($n \geq 0$).

a, work — `REAL` for `ssytrd`
`DOUBLE PRECISION` for `dsytrd`.
a(*lda*,*) is an array containing either upper or lower triangular part of the matrix A, as specified by *uplo*.
The second dimension of *a* must be at least max(1, *n*).

work(*lwork*) is a workspace array.

lda — `INTEGER`. The first dimension of *a*; at least max(1, *n*).

lwork — `INTEGER`. The size of the *work* array ($lwork \geq n$)
See *Application notes* for the suggested value of *lwork*.

Output Parameters

a — Overwritten by the tridiagonal matrix T and details of the orthogonal matrix Q, as specified by *uplo*.

d, *e*, *tau* REAL for ssytrd
DOUBLE PRECISION for dsytrd.
Arrays:
d(*) contains the diagonal elements of the matrix *T*.
The dimension of *d* must be at least max(1, *n*).

e(*) contains the off-diagonal elements of *T*.
The dimension of *e* must be at least max(1, *n*-1).

tau(*) stores further details of the orthogonal matrix *Q*. The dimension of *tau* must be at least max(1, *n*-1).

work(1) If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* =*n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The computed matrix T is exactly similar to a matrix $A + E$, where $\|E\|_2 = c(n)\varepsilon \|A\|_2$, $c(n)$ is a modestly increasing function of n, and ε is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

After calling this routine, you can call the following:

?orgtr to form the computed matrix *Q* explicitly;

?ormtr to multiply a real matrix by *Q*.

The complex counterpart of this routine is ?hetrd.

?orgtr

Generates the real orthogonal matrix Q determined by `?sytrd`.

Syntax

```
call sorgtr ( uplo, n, a, lda, tau, work, lwork, info )
call dorgtr ( uplo, n, a, lda, tau, work, lwork, info )
```

Description

The routine explicitly generates the *n* by *n* orthogonal matrix *Q* formed by `?sytrd` (see page 4-99) when reducing a real symmetric matrix *A* to tridiagonal form: $A = QTQ^T$. Use this routine after a call to `?sytrd`.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Use the same *uplo* as supplied to `?sytrd`.
n	`INTEGER`. The order of the matrix *Q* ($n \ge 0$).
a, tau, work	`REAL` for `sorgtr` `DOUBLE PRECISION` for `dorgtr`. Arrays: *a*(*lda*,*) is the array *a* as returned by `?sytrd`. The second dimension of *a* must be at least max(1, *n*). *tau*(*) is the array *tau* as returned by `?sytrd`. The dimension of *tau* must be at least max(1, *n*-1). *work* (*lwork*) is a workspace array.
lda	`INTEGER`. The first dimension of *a*; at least max(1, *n*).
lwork	`INTEGER`. The size of the *work* array ($lwork \ge n$) See *Application notes* for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the orthogonal matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* = (*n*-1)**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

The complex counterpart of this routine is ?ungtr.

?ormtr

Multiplies a real matrix by the real orthogonal matrix Q determined by `?sytrd`.

Syntax

```
call sormtr ( side,uplo,trans,m,n,a,lda,tau,c,ldc,work,lwork,info )
call dormtr ( side,uplo,trans,m,n,a,lda,tau,c,ldc,work,lwork,info )
```

Description

The routine multiplies a real matrix C by Q or Q^T, where Q is the orthogonal matrix Q formed by `?sytrd` (see page 4-99) when reducing a real symmetric matrix A to tridiagonal form: $A = QTQ^T$. Use this routine after a call to `?sytrd`.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products QC, Q^TC, CQ, or CQ^T (overwriting the result on C).

Input Parameters

In the descriptions below, *r* denotes the order of Q:
If *side* = 'L', *r* = *m*; if *side* = 'R', *r* = *n*.

side	`CHARACTER*1`. Must be either 'L' or 'R'. If *side* = 'L', Q or Q^T is applied to C from the left. If *side* = 'R', Q or Q^T is applied to C from the right.
uplo	`CHARACTER*1`. Must be 'U' or 'L'. Use the same *uplo* as supplied to `?sytrd`.
trans	`CHARACTER*1`. Must be either 'N' or 'T'. If *trans* = 'N', the routine multiplies C by Q. If *trans* = 'T', the routine multiplies C by Q^T.
m	`INTEGER`. The number of rows in the matrix C ($m \ge 0$).
n	`INTEGER`. The number of columns in C ($n \ge 0$).
a, *work*, *tau*, *c*	`REAL` for `sormtr` `DOUBLE PRECISION` for `dormtr`. *a*(*lda*,*) and *tau* are the arrays returned by `?sytrd`.

The second dimension of *a* must be at least max(1, *r*).
The dimension of *tau* must be at least max(1, *r*-1).

c(*ldc*,*) contains the matrix *C*.
The second dimension of *c* must be at least max(1, *n*)

work(*lwork*) is a workspace array.

lda INTEGER. The first dimension of *a*; *lda* ≥ max(1, *r*).

ldc INTEGER. The first dimension of *c*; *ldc* ≥ max(1, *n*).

lwork INTEGER. The size of the *work* array. Constraints:
lwork ≥ max(1, *n*) if *side* ='L';
lwork ≥ max(1, *m*) if *side* ='R'.
See *Application notes* for the suggested value of *lwork*.

Output Parameters

c Overwritten by the product QC, $Q^T C$, CQ, or CQ^T (as specified by *side* and *trans*).

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* = *n* * *blocksize* for *side* ='L', or *lwork* = *m* * *blocksize* for *side* ='R', where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The computed product differs from the exact product by a matrix E such that $||E||_2 = O(\varepsilon)\ ||C||_2$.

The total number of floating-point operations is approximately $2*m^2*n$ if *side* ='L' or $2*n^2*m$ if *side* ='R'.

The complex counterpart of this routine is ?unmtr.

?hetrd

Reduces a complex Hermitian matrix to tridiagonal form.

Syntax

```
call chetrd ( uplo,n,a,lda,d,e,tau,work,lwork,info )
call zhetrd ( uplo,n,a,lda,d,e,tau,work,lwork,info )
```

Description

This routine reduces a complex Hermitian matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = QTQ^H$. The unitary matrix Q is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided to work with Q in this representation. (They are described later in this section.)

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, *a* stores the upper triangular part of A.
If *uplo* = `'L'`, *a* stores the lower triangular part of A.

n — `INTEGER`. The order of the matrix A ($n \geq 0$).

a, work — `COMPLEX` for `chetrd`
`DOUBLE COMPLEX` for `zhetrd`.
a(*lda*,*) is an array containing either upper or lower triangular part of the matrix A, as specified by *uplo*.
The second dimension of *a* must be at least max(1, *n*).

work(*lwork*) is a workspace array.

lda — `INTEGER`. The first dimension of *a*; at least max(1, *n*).

lwork — `INTEGER`. The size of the *work* array ($lwork \geq n$)
See *Application notes* for the suggested value of *lwork*.

Output Parameters

a — Overwritten by the tridiagonal matrix T and details of the unitary matrix Q, as specified by *uplo*.

d, e — `REAL` for `chetrd`
`DOUBLE PRECISION` for `zhetrd`.
Arrays:
`d(*)` contains the diagonal elements of the matrix *T*.
The dimension of *d* must be at least max(1, *n*).

`e(*)` contains the off-diagonal elements of *T*.
The dimension of *e* must be at least max(1, *n*-1).

tau — `COMPLEX` for `chetrd`
`DOUBLE COMPLEX` for `zhetrd`.
Array, `DIMENSION` at least max(1, *n*-1).
Stores further details of the unitary matrix *Q*.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* =*n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The computed matrix *T* is exactly similar to a matrix $A + E$, where $\|E\|_2 = c(n)\varepsilon \|A\|_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

After calling this routine, you can call the following:

`?ungtr` — to form the computed matrix *Q* explicitly;

`?unmtr` — to multiply a complex matrix by *Q*.

The real counterpart of this routine is `?sytrd`.

?ungtr

Generates the complex unitary matrix Q determined by ?hetrd.

Syntax

```
call cungtr ( uplo, n, a, lda, tau, work, lwork, info )
call zungtr ( uplo, n, a, lda, tau, work, lwork, info )
```

Description

The routine explicitly generates the *n* by *n* unitary matrix *Q* formed by ?hetrd (see page 4-105) when reducing a complex Hermitian matrix *A* to tridiagonal form: $A = QTQ^H$. Use this routine after a call to ?hetrd.

Input Parameters

uplo — CHARACTER*1. Must be 'U' or 'L'.
Use the same *uplo* as supplied to ?hetrd.

n — INTEGER. The order of the matrix *Q* ($n \geq 0$).

a, tau, work — COMPLEX for cungtr
DOUBLE COMPLEX for zungtr.
Arrays:
a(*lda*,*) is the array *a* as returned by ?hetrd.
The second dimension of *a* must be at least max(1, *n*).

tau(*) is the array *tau* as returned by ?hetrd.
The dimension of *tau* must be at least max(1, *n*-1).

work (*lwork*) is a workspace array.

lda — INTEGER. The first dimension of *a*; at least max(1, *n*).

lwork — INTEGER. The size of the *work* array ($lwork \geq n$)
See *Application notes* for the suggested value of *lwork*.

Output Parameters

a — Overwritten by the unitary matrix *Q*.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* = (*n*-1)**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The computed matrix Q differs from an exactly unitary matrix by a matrix E such that $||E||_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

The real counterpart of this routine is ?orgtr.

?unmtr

Multiplies a complex matrix by the complex unitary matrix Q determined by ?hetrd.

Syntax

```
call cunmtr ( side,uplo,trans,m,n,a,lda,tau,c,ldc,work,lwork,info )
call zunmtr ( side,uplo,trans,m,n,a,lda,tau,c,ldc,work,lwork,info )
```

Description

The routine multiplies a complex matrix C by Q or Q^H, where Q is the unitary matrix Q formed by ?hetrd (see page 4-105) when reducing a complex Hermitian matrix A to tridiagonal form: $A = QTQ^H$. Use this routine after a call to ?hetrd.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products QC, Q^HC, CQ, or CQ^H (overwriting the result on C).

Input Parameters

In the descriptions below, *r* denotes the order of Q:
If *side* = 'L', *r* = *m*; if *side* = 'R', *r* = *n*.

side	CHARACTER*1. Must be either 'L' or 'R'. If *side* = 'L', Q or Q^H is applied to C from the left. If *side* = 'R', Q or Q^H is applied to C from the right.
uplo	CHARACTER*1. Must be 'U' or 'L'. Use the same *uplo* as supplied to ?hetrd.
trans	CHARACTER*1. Must be either 'N' or 'T'. If *trans* = 'N', the routine multiplies C by Q. If *trans* = 'T', the routine multiplies C by Q^H.
m	INTEGER. The number of rows in the matrix C ($m \ge 0$).
n	INTEGER. The number of columns in C ($n \ge 0$).
a, work, tau, c	COMPLEX for cunmtr DOUBLE COMPLEX for zunmtr. *a*(*lda*,*) and *tau* are the arrays returned by ?hetrd.

The second dimension of `a` must be at least max(1, `r`).
The dimension of `tau` must be at least max(1, `r`-1).

`c(ldc,*)` contains the matrix *C*.
The second dimension of `c` must be at least max(1, `n`)

`work(lwork)` is a workspace array.

`lda` — `INTEGER`. The first dimension of `a`; `lda` ≥ max(1, `r`).

`ldc` — `INTEGER`. The first dimension of `c`; `ldc` ≥ max(1, `n`).

`lwork` — `INTEGER`. The size of the `work` array. Constraints:
`lwork` ≥ max(1, `n`) if `side` ='L';
`lwork` ≥ max(1, `m`) if `side` ='R'.
See *Application notes* for the suggested value of `lwork`.

Output Parameters

`c` — Overwritten by the product *QC*, $Q^H C$, *CQ*, or CQ^H (as specified by `side` and `trans`).

`work(1)` — If `info` = 0, on exit `work(1)` contains the minimum value of `lwork` required for optimum performance. Use this `lwork` for subsequent runs.

`info` — `INTEGER`.
If `info` = 0, the execution is successful.
If `info` = -`i`, the `i`th parameter had an illegal value.

Application Notes

For better performance, try using `lwork` = `n`**blocksize* (for `side` ='L') or `lwork` = `m`**blocksize* (for `side` ='R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The computed product differs from the exact product by a matrix *E* such that $\|E\|_2 = O(\varepsilon)\,\|C\|_2$, where ε is the machine precision.

The total number of floating-point operations is approximately $8*m^2*n$ if `side` ='L' or $8*n^2*m$ if `side` ='R'.

The real counterpart of this routine is `?ormtr`.

?sptrd

Reduces a real symmetric matrix to tridiagonal form using packed storage.

Syntax

```
call ssptrd ( uplo,n,ap,d,e,tau,info )
call dsptrd ( uplo,n,ap,d,e,tau,info )
```

Description

This routine reduces a packed real symmetric matrix *A* to symmetric tridiagonal form *T* by an orthogonal similarity transformation: $A = QTQ^T$. The orthogonal matrix *Q* is not formed explicitly but is represented as a product of `n-1` elementary reflectors. Routines are provided for working with *Q* in this representation. (They are described later in this section.)

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. If *uplo* = `'U'`, *ap* stores the packed upper triangle of *A*. If *uplo* = `'L'`, *ap* stores the packed lower triangle of *A*.
n	`INTEGER`. The order of the matrix *A* ($n \geq 0$).
ap	`REAL` for `ssptrd` `DOUBLE PRECISION` for `dsptrd`. Array, `DIMENSION` at least max(1,*n*(*n*+1)/2). Contains either upper or lower triangle of *A* (as specified by *uplo*) in packed form.

Output Parameters

ap	Overwritten by the tridiagonal matrix *T* and details of the orthogonal matrix *Q*, as specified by *uplo*.
d, e, tau	`REAL` for `ssptrd` `DOUBLE PRECISION` for `dsptrd`. Arrays: *d*(*) contains the diagonal elements of the matrix *T*. The dimension of *d* must be at least max(1, *n*).

	`e(*)` contains the off-diagonal elements of *T*. The dimension of `e` must be at least max(1, `n`-1). `tau(*)` stores further details of the matrix *Q*. The dimension of `tau` must be at least max(1, `n`-1).
info	`INTEGER`. If `info` = 0, the execution is successful. If `info` = `-i`, the `i`th parameter had an illegal value.

Application Notes

The computed matrix *T* is exactly similar to a matrix *A* + *E*, where $||E||_2 = c(n)\varepsilon\, ||A||_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

After calling this routine, you can call the following:

`?opgtr`	to form the computed matrix *Q* explicitly;
`?opmtr`	to multiply a real matrix by *Q*.

The complex counterpart of this routine is `?hptrd`.

?opgtr

Generates the real orthogonal matrix Q determined by `?sptrd`.

Syntax

```
call sopgtr ( uplo, n, ap, tau, q, ldq, work, info )
call dopgtr ( uplo, n, ap, tau, q, ldq, work, info )
```

Description

The routine explicitly generates the *n* by *n* orthogonal matrix *Q* formed by `?sptrd` (see page 4-111) when reducing a packed real symmetric matrix *A* to tridiagonal form: $A = QTQ^T$. Use this routine after a call to `?sptrd`.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Use the same *uplo* as supplied to `?sptrd`.
n	`INTEGER`. The order of the matrix *Q* ($n \ge 0$).
ap, tau	`REAL` for `sopgtr` `DOUBLE PRECISION` for `dopgtr`. Arrays *ap* and *tau*, as returned by `?sptrd`. The dimension of *ap* must be at least max(1, *n*(*n*+1)/2). The dimension of *tau* must be at least max(1, *n*-1).
ldq	`INTEGER`. The first dimension of the output array *q*; at least max(1, *n*).
work	`REAL` for `sopgtr` `DOUBLE PRECISION` for `dopgtr`. Workspace array, `DIMENSION` at least max(1, *n*-1).

Output Parameters

q	`REAL` for `sopgtr` `DOUBLE PRECISION` for `dopgtr`. Array, `DIMENSION (`*ldq*`,*)`. Contains the computed matrix *Q*. The second dimension of *q* must be at least max(1, *n*).

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

The complex counterpart of this routine is ?upgtr.

?opmtr

Multiplies a real matrix by the real orthogonal matrix Q determined by ?sptrd.

Syntax

```
call sopmtr (side,uplo,trans,m,n,ap,tau,c,ldc,work,info)
call dopmtr (side,uplo,trans,m,n,ap,tau,c,ldc,work,info)
```

Description

The routine multiplies a real matrix C by Q or Q^T, where Q is the orthogonal matrix Q formed by ?sptrd (see page 4-111) when reducing a packed real symmetric matrix A to tridiagonal form: $A = QTQ^T$. Use this routine after a call to ?sptrd.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products QC, Q^TC, CQ, or CQ^T (overwriting the result on C).

Input Parameters

In the descriptions below, *r* denotes the order of Q:
If *side* = 'L', $r = m$; if *side* = 'R', $r = n$.

side — CHARACTER*1. Must be either 'L' or 'R'.
If *side* = 'L', Q or Q^T is applied to C from the left.
If *side* = 'R', Q or Q^T is applied to C from the right.

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Use the same *uplo* as supplied to `?sptrd`.
trans	`CHARACTER*1`. Must be either `'N'` or `'T'`. If *trans* = `'N'`, the routine multiplies C by Q. If *trans* = `'T'`, the routine multiplies C by Q^T.
m	`INTEGER`. The number of rows in the matrix C ($m \geq 0$).
n	`INTEGER`. The number of columns in C ($n \geq 0$).
ap, *work*, *tau*, *c*	`REAL` for `sopmtr` `DOUBLE PRECISION` for `dopmtr`. *ap* and *tau* are the arrays returned by `?sptrd`. The dimension of *ap* must be at least max(1, $r(r+1)/2$). The dimension of *tau* must be at least max(1, $r-1$). *c*(*ldc*,*) contains the matrix C. The second dimension of *c* must be at least max(1, *n*) *work*(*) is a workspace array. The dimension of *work* must be at least max(1, *n*) if *side* = `'L'`; max(1, *m*) if *side* = `'R'`.
ldc	`INTEGER`. The first dimension of *c*; *ldc* ≥ max(1, *n*).

Output Parameters

c	Overwritten by the product QC, Q^TC, CQ, or CQ^T (as specified by *side* and *trans*).
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The computed product differs from the exact product by a matrix E such that $\|E\|_2 = O(\varepsilon)\,\|C\|_2$, where ε is the machine precision.

The total number of floating-point operations is approximately $2*m^2*n$ if *side* = `'L'` or $2*n^2*m$ if *side* = `'R'`.

The complex counterpart of this routine is `?upmtr`.

?hptrd

Reduces a complex Hermitian matrix to tridiagonal form using packed storage.

Syntax

```
call chptrd ( uplo,n,ap,d,e,tau,info )
call zhptrd ( uplo,n,ap,d,e,tau,info )
```

Description

This routine reduces a packed complex Hermitian matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = QTQ^H$. The unitary matrix Q is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided for working with Q in this representation. (They are described later in this section.)

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* =`'U'`, *ap* stores the packed upper triangle of A.
If *uplo* =`'L'`, *ap* stores the packed lower triangle of A.

n — `INTEGER`. The order of the matrix A ($n \geq 0$).

ap — `COMPLEX` for `chptrd`
`DOUBLE COMPLEX` for `zhptrd`.
Array, `DIMENSION` at least max(1,*n*(*n*+1)/2).
Contains either upper or lower triangle of A (as specified by *uplo*) in packed form.

Output Parameters

ap — Overwritten by the tridiagonal matrix T and details of the orthogonal matrix Q, as specified by *uplo*.

d, e — `REAL` for `chptrd`
`DOUBLE PRECISION` for `zhptrd`.
Arrays:
d(*) contains the diagonal elements of the matrix T.
The dimension of *d* must be at least max(1, *n*).

	`e(*)` contains the off-diagonal elements of *T*. The dimension of `e` must be at least max(1, `n-1`).
tau	`COMPLEX` for `chptrd` `DOUBLE COMPLEX` for `zhptrd`. Arrays, DIMENSION at least max(1, `n-1`). Contains further details of the orthogonal matrix *Q*.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

The computed matrix *T* is exactly similar to a matrix $A + E$, where $||E||_2 = c(n)\varepsilon\ ||A||_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

After calling this routine, you can call the following:

`?upgtr`	to form the computed matrix *Q* explicitly;
`?upmtr`	to multiply a complex matrix by *Q*.

The real counterpart of this routine is `?sptrd`.

?upgtr

Generates the complex unitary matrix Q determined by `?hptrd`.

Syntax

```
call cupgtr ( uplo, n, ap, tau, q, ldq, work, info )
call zupgtr ( uplo, n, ap, tau, q, ldq, work, info )
```

Description

The routine explicitly generates the *n* by *n* unitary matrix Q formed by `?hptrd` (see page 4-116) when reducing a packed complex Hermitian matrix A to tridiagonal form: $A = QTQ^H$. Use this routine after a call to `?hptrd`.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Use the same *uplo* as supplied to `?sptrd`.
n	`INTEGER`. The order of the matrix Q ($n \geq 0$).
ap, tau	`COMPLEX` for `cupgtr` `DOUBLE COMPLEX` for `zupgtr`. Arrays *ap* and *tau*, as returned by `?hptrd`. The dimension of *ap* must be at least max(1, *n*(*n*+1)/2). The dimension of *tau* must be at least max(1, *n*-1).
ldq	`INTEGER`. The first dimension of the output array *q*; at least max(1, *n*).
work	`COMPLEX` for `cupgtr` `DOUBLE COMPLEX` for `zupgtr`. Workspace array, `DIMENSION` at least max(1, *n*-1).

Output Parameters

q	`COMPLEX` for `cupgtr` `DOUBLE COMPLEX` for `zupgtr`. Array, `DIMENSION (`*ldq*`,*)`. Contains the computed matrix Q. The second dimension of *q* must be at least max(1, *n*).

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

The real counterpart of this routine is ?opgtr.

?upmtr

Multiplies a complex matrix by the unitary matrix Q determined by ?hptrd.

Syntax

```
call cupmtr (side,uplo,trans,m,n,ap,tau,c,ldc,work,info)
call zupmtr (side,uplo,trans,m,n,ap,tau,c,ldc,work,info)
```

Description

The routine multiplies a complex matrix C by Q or Q^H, where Q is the unitary matrix Q formed by ?hptrd (see page 4-116) when reducing a packed complex Hermitian matrix A to tridiagonal form: $A = QTQ^H$. Use this routine after a call to ?hptrd.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products QC, Q^HC, CQ, or CQ^H (overwriting the result on C).

Input Parameters

In the descriptions below, *r* denotes the order of Q:
If *side* = 'L', *r* = *m*; if *side* = 'R', *r* = *n*.

side CHARACTER*1. Must be either 'L' or 'R'.
If *side* = 'L', Q or Q^H is applied to C from the left.
If *side* = 'R', Q or Q^H is applied to C from the right.

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Use the same *uplo* as supplied to `?hptrd`.
trans	`CHARACTER*1`. Must be either `'N'` or `'T'`. If *trans* = `'N'`, the routine multiplies C by Q. If *trans* = `'T'`, the routine multiplies C by Q^H.
m	`INTEGER`. The number of rows in the matrix C ($m \geq 0$).
n	`INTEGER`. The number of columns in C ($n \geq 0$).
ap, tau, c, work	`COMPLEX` for `cupmtr` `DOUBLE COMPLEX` for `zupmtr`. *ap* and *tau* are the arrays returned by `?hptrd`. The dimension of *ap* must be at least max(1, $r(r+1)/2$). The dimension of *tau* must be at least max(1, $r-1$). *c(ldc,*)* contains the matrix C. The second dimension of *c* must be at least max(1, *n*) *work(*)* is a workspace array. The dimension of *work* must be at least max(1, *n*) if *side* = `'L'`; max(1, *m*) if *side* = `'R'`.
ldc	`INTEGER`. The first dimension of *c*; *ldc* ≥ max(1, *n*).

Output Parameters

c	Overwritten by the product QC, Q^HC, CQ, or CQ^H (as specified by *side* and *trans*).
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

The computed product differs from the exact product by a matrix E such that $\|E\|_2 = O(\varepsilon)\|C\|_2$, where ε is the machine precision.

The total number of floating-point operations is approximately $8*m^2*n$ if *side* = `'L'` or $8*n^2*m$ if *side* = `'R'`.

The real counterpart of this routine is `?opmtr`.

?sbtrd

Reduces a real symmetric band matrix to tridiagonal form.

Syntax

```
call ssbtrd (vect,uplo,n,kd,ab,ldab,d,e,q,ldq,work,info)
call dsbtrd (vect,uplo,n,kd,ab,ldab,d,e,q,ldq,work,info)
```

Description

This routine reduces a real symmetric band matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = QTQ^T$. The orthogonal matrix Q is determined as a product of Givens rotations. If required, the routine can also form the matrix Q explicitly.

Input Parameters

vect — `CHARACTER*1`. Must be `'V'` or `'N'`.
If *vect* = `'V'`, the routine returns the explicit matrix Q.
If *vect* = `'N'`, the routine does not return Q.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, *ab* stores the upper triangular part of A.
If *uplo* = `'L'`, *ab* stores the lower triangular part of A.

n — `INTEGER`. The order of the matrix A ($n \ge 0$).

kd — `INTEGER`. The number of super- or sub-diagonals in A ($kd \ge 0$).

ab, work — `REAL` for `ssbtrd`
`DOUBLE PRECISION` for `dsbtrd`.
ab (*ldab*, *) is an array containing either upper or lower triangular part of the matrix A (as specified by *uplo*) in band storage format.
The second dimension of *ab* must be at least max(1, *n*).

work (*) is a workspace array.
The dimension of *work* must be at least max(1, *n*).

ldab — `INTEGER`. The first dimension of *ab*; at least *kd*+1.

ldq — `INTEGER`. The first dimension of *q*. Constraints:
$ldq \ge \max(1, n)$ if *vect* = `'V'`;
$ldq \ge 1$ if *vect* = `'N'`.

Output Parameters

ab	On exit, the array *ab* is overwritten.
d, e, q	REAL for ssbtrd DOUBLE PRECISION for dsbtrd. Arrays: *d*(*) contains the diagonal elements of the matrix *T*. The dimension of *d* must be at least max(1, *n*). *e*(*) contains the off-diagonal elements of *T*. The dimension of *e* must be at least max(1, *n*-1). *q*(*ldq*,*) is not referenced if *vect* = 'N'. If *vect* ='V', *q* contains the *n* by *n* matrix *Q*. The second dimension of *q* must be: at least max(1, *n*) if *vect* = 'V'; at least 1 if *vect* = 'N'.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The computed matrix T is exactly similar to a matrix $A + E$, where $||E||_2 = c(n)\varepsilon\,||A||_2$, $c(n)$ is a modestly increasing function of n, and ε is the machine precision. The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $||E||_2 = O(\varepsilon)$.

The total number of floating-point operations is approximately $6n^2*kd$ if *vect* ='N', with $3n^3*(kd-1)/kd$ additional operations if *vect* ='V'.

The complex counterpart of this routine is ?hbtrd.

?hbtrd

Reduces a complex Hermitian band matrix to tridiagonal form.

Syntax

```
call chbtrd (vect,uplo,n,kd,ab,ldab,d,e,q,ldq,work,info)
call zhbtrd (vect,uplo,n,kd,ab,ldab,d,e,q,ldq,work,info)
```

Description

This routine reduces a complex Hermitian band matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = QTQ^H$. The unitary matrix Q is determined as a product of Givens rotations. If required, the routine can also form the matrix Q explicitly.

Input Parameters

vect — `CHARACTER*1`. Must be `'V'` or `'N'`.
If *vect* = `'V'`, the routine returns the explicit matrix Q.
If *vect* = `'N'`, the routine does not return Q.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, *ab* stores the upper triangular part of A.
If *uplo* = `'L'`, *ab* stores the lower triangular part of A.

n — `INTEGER`. The order of the matrix A ($n \geq 0$).

kd — `INTEGER`. The number of super- or sub-diagonals in A ($kd \geq 0$).

ab, work — `COMPLEX` for `chbtrd`
`DOUBLE COMPLEX` for `zhbtrd`.
ab (*ldab*,*) is an array containing either upper or lower triangular part of the matrix A (as specified by *uplo*) in band storage format.
The second dimension of *ab* must be at least max(1, *n*).

work (*) is a workspace array.
The dimension of *work* must be at least max(1, *n*).

ldab — `INTEGER`. The first dimension of *ab*; at least *kd*+1.

ldq — `INTEGER`. The first dimension of *q*. Constraints:
$ldq \geq \max(1, n)$ if *vect* = `'V'`;
$ldq \geq 1$ if *vect* = `'N'`.

Output Parameters

ab	On exit, the array *ab* is overwritten.
d, e	`REAL` for `chbtrd` `DOUBLE PRECISION` for `zhbtrd`. Arrays: `d(*)` contains the diagonal elements of the matrix *T*. The dimension of *d* must be at least max(1, *n*). `e(*)` contains the off-diagonal elements of *T*. The dimension of *e* must be at least max(1, *n*-1).
q	`COMPLEX` for `chbtrd` `DOUBLE COMPLEX` for `zhbtrd`. Array, `DIMENSION` (*ldq*,*). If *vect* ='N', *q* is not referenced. If *vect* ='V', *q* contains the *n* by *n* matrix *Q*. The second dimension of *q* must be: at least max(1, *n*) if *vect* = 'V'; at least 1 if *vect* = 'N'.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The computed matrix *T* is exactly similar to a matrix *A* + *E*, where $||E||_2 = c(n)\varepsilon\, ||A||_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision. The computed matrix *Q* differs from an exactly unitary matrix by a matrix *E* such that $||E||_2 = O(\varepsilon)$.

The total number of floating-point operations is approximately $20n^2*kd$ if *vect* ='N', with $10n^3*(kd-1)/kd$ additional operations if *vect* ='V'.

The real counterpart of this routine is `?sbtrd`.

?sterf

Computes all eigenvalues of a real symmetric tridiagonal matrix using QR algorithm.

Syntax

```
call ssterf ( n, d, e, info )
call dsterf ( n, d, e, info )
```

Description

This routine computes all the eigenvalues of a real symmetric tridiagonal matrix T (which can be obtained by reducing a symmetric or Hermitian matrix to tridiagonal form). The routine uses a square-root-free variant of the QR algorithm.

If you need not only the eigenvalues but also the eigenvectors, call `?steqr` (page 4-127).

Input Parameters

n — `INTEGER`. The order of the matrix T ($n \ge 0$).

d, e — `REAL` for `ssterf`
`DOUBLE PRECISION` for `dsterf`.
Arrays:
d(*) contains the diagonal elements of T.
The dimension of *d* must be at least max(1, *n*).

e(*) contains the off-diagonal elements of T.
The dimension of *e* must be at least max(1, *n*-1).

Output Parameters

d — The *n* eigenvalues in ascending order, unless *info* > 0.
See also *info*.

e — On exit, the array is overwritten; see *info*.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = *i*, the algorithm failed to find all the eigenvalues after 30*n* iterations: *i* off-diagonal elements have not converged to zero. On exit, *d* and *e* contain,

respectively, the diagonal and off-diagonal elements of a tridiagonal matrix orthogonally similar to *T*.
If `info` = `-i`, the `i`th parameter had an illegal value.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $||E||_2 = O(\varepsilon) ||T||_2$, where ε is the machine precision.

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then

$$|\mu_i - \lambda_i| \le c(n)\varepsilon \, ||T||_2$$

where $c(n)$ is a modestly increasing function of n.

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about $14n^2$.

?steqr

Computes all eigenvalues and eigenvectors of a symmetric or Hermitian matrix reduced to tridiagonal form (QR algorithm).

Syntax

```
call ssteqr ( compz, n, d, e, z, ldz, work, info )
call dsteqr ( compz, n, d, e, z, ldz, work, info )
call csteqr ( compz, n, d, e, z, ldz, work, info )
call zsteqr ( compz, n, d, e, z, ldz, work, info )
```

Description

This routine computes all the eigenvalues and (optionally) all the eigenvectors of a real symmetric tridiagonal matrix T. In other words, the routine can compute the spectral factorization: $T = Z\Lambda Z^T$. Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i; Z is an orthogonal matrix whose columns are eigenvectors. Thus,

$$Tz_i = \lambda_i z_i \text{ for } i = 1, 2, \ldots, n.$$

(The routine normalizes the eigenvectors so that $\|z_i\|_2 = 1$.)

You can also use the routine for computing the eigenvalues and eigenvectors of an arbitrary real symmetric (or complex Hermitian) matrix A reduced to tridiagonal form T: $A = QTQ^H$. In this case, the spectral factorization is as follows: $A = QTQ^H = (QZ)\Lambda(QZ)^H$. Before calling `?steqr`, you must reduce A to tridiagonal form and generate the explicit matrix Q by calling the following routines:

	for real matrices:	for complex matrices:
full storage	`?sytrd,?orgtr`	`?hetrd,?ungtr`
packed storage	`?sptrd,?opgtr`	`?hptrd,?upgtr`
band storage	`?sbtrd (vect='V')`	`?hbtrd (vect='V')`

If you need eigenvalues only, it's more efficient to call `?sterf` (page 4-125). If T is positive-definite, `?pteqr` (page 4-137) can compute small eigenvalues more accurately than `?steqr`.

To solve the problem by a single call, use one of the divide and conquer routines `?stevd`, `?syevd`, `?spevd`, or `?sbevd` for real symmetric matrices or `?heevd`, `?hpevd`, or `?hbevd` for complex Hermitian matrices.

Input Parameters

compz
: CHARACTER*1. Must be 'N' or 'I' or 'V'.
If *compz* ='N', the routine computes eigenvalues only.
If *compz* ='I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix *T*.
If *compz* ='V', the routine computes the eigenvalues and eigenvectors of *A* (and the array *z* must contain the matrix *Q* on entry).

n
: INTEGER. The order of the matrix *T* ($n \geq 0$).

d, *e*, *work*
: REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Arrays:
d(*) contains the diagonal elements of *T*.
The dimension of *d* must be at least max(1, *n*).

e(*) contains the off-diagonal elements of *T*.
The dimension of *e* must be at least max(1, *n*-1).

work(*) is a workspace array.
The dimension of *work* must be:
at least 1 if *compz* = 'N';
at least max(1, 2*n-2) if *compz* ='V' or 'I'.

z
: REAL for ssteqr
DOUBLE PRECISION for dsteqr
COMPLEX for csteqr
DOUBLE COMPLEX for zsteqr.
Array, DIMENSION (*ldz*, *)
If *compz* ='N' or 'I', *z* need not be set.
If *vect* ='V', *z* must contain the *n* by *n* matrix *Q*.
The second dimension of *z* must be:
at least 1 if *compz* = 'N';
at least max(1, *n*) if *compz* ='V' or 'I'.

work (*lwork*) is a workspace array.

ldz
: INTEGER. The first dimension of *z*. Constraints:
ldz ≥ 1 if *compz* = 'N';
ldz ≥ max(1, *n*) if *compz* ='V' or 'I'.

Output Parameters

d The *n* eigenvalues in ascending order, unless *info* > 0. See also *info*.

e On exit, the array is overwritten; see *info*.

z If *info* = 0, contains the *n* orthonormal eigenvectors, stored by columns. (The *i*th column corresponds to the *i*th eigenvalue.)

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = *i*, the algorithm failed to find all the eigenvalues after 30*n* iterations: *i* off-diagonal elements have not converged to zero. On exit, *d* and *e* contain, respectively, the diagonal and off-diagonal elements of a tridiagonal matrix orthogonally similar to *T*.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $\|E\|_2 = O(\varepsilon)\|T\|_2$, where ε is the machine precision.

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then

$$|\mu_i - \lambda_i| \le c(n)\varepsilon \|T\|_2$$

where $c(n)$ is a modestly increasing function of n.

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$$\theta(z_i, w_i) \le c(n)\varepsilon \|T\|_2 / \min_{i \ne j}|\lambda_i - \lambda_j|.$$

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about

$24n^2$ if *compz* = 'N';
$7n^3$ (for complex flavors, $14n^3$) if *compz* = 'V' or 'I'.

?stedc

Computes all eigenvalues and eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.

Syntax

```
call sstedc(compz, n, d, e, z, ldz, work, lwork, iwork, liwork,info)
call dstedc(compz, n, d, e, z, ldz, work, lwork, iwork, liwork,info)
call cstedc(compz, n, d, e, z, ldz, work, lwork, rwork, lrwork,
            iwork, liwork,info)
call zstedc(compz, n, d, e, z, ldz, work, lwork, rwork, lrwork,
            iwork, liwork,info)
```

Description

This routine computes all the eigenvalues and (optionally) all the eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.
The eigenvectors of a full or band real symmetric or complex Hermitian matrix can also be found if `?sytrd/?hetrd` or `?sptrd/?hptrd` or `?sbtrd/?hbtrd` has been used to reduce this matrix to tridiagonal form.

Input Parameters

compz — `CHARACTER*1`. Must be `'N'` or `'I'` or `'V'`.
If *compz* = `'N'`, the routine computes eigenvalues only.
If *compz* = `'I'`, the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix.
If *compz* = `'V'`, the routine computes the eigenvalues and eigenvectors of original symmetric/Hermitian matrix. On entry, the array *z* must contain the orthogonal/unitary matrix used to reduce the original matrix to tridiagonal form.

n — `INTEGER`. The order of the symmetric tridiagonal matrix ($n \geq 0$).

d, e, rwork — `REAL` for single-precision flavors
`DOUBLE PRECISION` for double-precision flavors.
Arrays:
d(*) contains the diagonal elements of the tridiagonal matrix. The dimension of *d* must be at least max(1, *n*).

	`e(*)` contains the subdiagonal elements of the tridiagonal matrix. The dimension of *e* must be at least max(1, *n*-1). *rwork*(*lrwork*) is a workspace array used in complex flavors only.
z, work	REAL for sstedc DOUBLE PRECISION for dstedc COMPLEX for cstedc DOUBLE COMPLEX for zstedc. Arrays: *z*(*ldz*, *), *work*(*). If *compz* ='V', then, on entry, *z* must contain the orthogonal/unitary matrix used to reduce the original matrix to tridiagonal form. The second dimension of *z* must be at least max(1, *n*). *work* (*lwork*) is a workspace array.
ldz	INTEGER. The first dimension of *z*. Constraints: *ldz* ≥ 1 if *compz* = 'N'; *ldz* ≥ max(1, *n*) if *compz* ='V' or 'I'.
lwork	INTEGER. The dimension of the array *work*. See *Application Notes* for the required value of *lwork*.
lrwork	INTEGER. The dimension of the array *rwork* (used for complex flavors only). See *Application Notes* for the required value of *lrwork*.
iwork	INTEGER. Workspace array, DIMENSION (*liwork*).
liwork	INTEGER. The dimension of the array *iwork*. See *Application Notes* for the required value of *liwork*.

Output Parameters

d	The *n* eigenvalues in ascending order, unless *info* ≠ 0. See also *info*.
e	On exit, the array is overwritten; see *info*.
z	If *info* = 0, then if *compz* ='V', *z* contains the orthonormal eigenvectors of the original symmetric/Hermitian matrix, and if *compz* ='I', *z* contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If *compz* ='N', *z* is not referenced.
work(1)	On exit, if *info* = 0, then *work(1)* returns the optimal *lwork*.
rwork(1)	On exit, if *info* = 0, then *rwork(1)* returns the optimal *lrwork* (for complex flavors only).

iwork(1)	On exit, if *info* = 0, then *iwork(1)* returns the optimal *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *-i*, the *i*th parameter had an illegal value.If *info* = *i*, the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *i*/(n+1) through mod(*i*, n+1).

Application Notes

The required size of workspace arrays must be as follows.

For `sstedc/dstedc`:

If *compz* = 'N' or $n \le 1$ then *lwork* must be at least 1.
If *compz* = 'V' and $n > 1$ then *lwork* must be at least
$(1 + 3n + 2n \cdot \lg n + 3n^2)$, where lg(n) = smallest integer k such that $2^k \ge n$.

If *compz* = 'I' and $n > 1$ then *lwork* must be at least $(1 + 4n + n^2)$.

If *compz* = 'N' or $n \le 1$ then *liwork* must be at least 1.
If *compz* = 'V' and $n > 1$ then *liwork* must be at least $(6 + 6n + 5n \cdot \lg n)$.
If *compz* = 'I' and $n > 1$ then *liwork* must be at least $(3 + 5n)$.

For `cstedc/zstedc`:

If *compz* = 'N' or 'I', or $n \le 1$, *lwork* must be at least 1.
If *compz* = 'V' and $n > 1$, *lwork* must be at least n^2.

If *compz* = 'N' or $n \le 1$, *lrwork* must be at least 1.
If *compz* = 'V' and $n > 1$, *lrwork* must be at least
$(1 + 3n + 2n \cdot \lg n + 3n^2)$, where lg(n) = smallest integer k such that $2^k \ge n$.

If *compz* = 'I' and $n > 1$, *lrwork* must be at least$(1 + 4n + 2n^2)$.

The required value of *liwork* for complex flavors is the same as for real flavors.

?stegr

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

```
call sstegr (jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z,
             ldz, isuppz, work, lwork, iwork, liwork, info)
call dstegr (jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z,
             ldz, isuppz, work, lwork, iwork, liwork, info)
call cstegr (jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z,
             ldz, isuppz, work, lwork, iwork, liwork, info)
call zstegr (jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z,
             ldz, isuppz, work, lwork, iwork, liwork, info)
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix *T*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues. The eigenvalues are computed by the *dqds* algorithm, while orthogonal eigenvectors are computed from various "good" LDL^T representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the i-th unreduced block of *T*,

(a) Compute $T - \sigma_i = L_i D_i L_i^T$, such that $L_i D_i L_i^T$ is a relatively robust representation;

(b) Compute the eigenvalues, λ_j, of $L_i D_i L_i^T$ to high relative accuracy by the *dqds* algorithm;

(c) If there is a cluster of close eigenvalues, "choose" σ_i close to the cluster, and go to step (a);

(d) Given the approximate eigenvalue λ_j of $L_i D_i L_i^T$, compute the corresponding eigenvector by forming a rank-revealing twisted factorization.

The desired accuracy of the output can be specified by the input parameter `abstol`.

Input Parameters

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *job* =`'N'`, then only eigenvalues are computed.
If *job* =`'V'`, then eigenvalues and eigenvectors are computed.

range — `CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`.
If *range* =`'A'`, the routine computes all eigenvalues.
If *range* =`'V'`, the routine computes eigenvalues λ_i in the half-open interval: $vl < \lambda_i \le vu$.
If *range* =`'I'`, the routine computes eigenvalues with indices *il* to *iu*.

n — `INTEGER`. The order of the matrix T ($n \ge 0$).

d, e, work — `REAL` for single precision flavors
`DOUBLE PRECISION` for double precision flavors.
Arrays:
d(*) contains the diagonal elements of T.
The dimension of *d* must be at least max(1, *n*).

e(*) contains the subdiagonal elements of T in elements 1 to *n*-1; *e*(*n*) need not be set.
The dimension of *e* must be at least max(1, *n*).

work(*lwork*) is a workspace array.

vl, vu — `REAL` for single precision flavors
`DOUBLE PRECISION` for double precision flavors.
If *range* =`'V'`, the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: $vl < vu$.

If *range* =`'A'` or `'I'`, *vl* and *vu* are not referenced.

il, iu — `INTEGER`.
If *range* =`'I'`, the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: $1 \le il \le iu \le n$, if $n > 0$; *il*=1 and *iu*=0 if $n = 0$.

If *range* =`'A'` or `'V'`, *il* and *iu* are not referenced.

abstol — `REAL` for single precision flavors
`DOUBLE PRECISION` for double precision flavors.
The absolute tolerance to which each eigenvalue/eigenvector is required.
If *jobz* = `'V'`, the eigenvalues and eigenvectors output have residual norms

bounded by *abstol*, and the dot products between different eigenvectors are bounded by *abstol*. If $abstol < n\varepsilon\|T\|_1$, then $n\varepsilon\|T\|_1$ will be used in its place, where ε is the machine precision. The eigenvalues are computed to an accuracy of $\varepsilon\|T\|_1$ irrespective of *abstol*. If high relative accuracy is important, set *abstol* to `?lamch` ('Safe minimum').

ldz — `INTEGER`. The leading dimension of the output array *z*. Constraints:
$ldz \geq 1$ if *jobz* = 'N';
$ldz \geq \max(1, n)$ if *jobz* = 'V'.

lwork — `INTEGER`. The dimension of the array *work*,
$lwork \geq \max(1, 18n)$.

iwork — `INTEGER`.
Workspace array, `DIMENSION` (*liwork*).

liwork — `INTEGER`. The dimension of the array *iwork*,
$lwork \geq \max(1, 10n)$.

Output Parameters

d, e — On exit, *d* and *e* are overwritten.

m — `INTEGER`. The total number of eigenvalues found,
$0 \leq m \leq n$. If *range* = 'A', $m = n$, and if *range* = 'I',
$m = iu - il + 1$.

w — `REAL` for single precision flavors
`DOUBLE PRECISION` for double precision flavors.
Array, `DIMENSION` at least max(1, *n*).
The selected eigenvalues in ascending order, stored in *w*(1) to *w*(*m*).

z — `REAL` for `sstegr`
`DOUBLE PRECISION` for `dstegr`
`COMPLEX` for `cstegr`
`DOUBLE COMPLEX` for `zstegr`.
Array *z*(*ldz*, *), the second dimension of *z* must be at least max(1, *m*).

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). If *jobz* = 'N', then *z* is not referenced.
Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z* ; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

isuppz	`INTEGER`. Array, `DIMENSION` at least 2*max(1, *m*). The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th eigenvector is nonzero only in elements *isuppz*(2*i*-1) through *isuppz*(2*i*).
work(1)	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
iwork(1)	On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = *-i*, the *i*th parameter had an illegal value. If *info* = 1, internal error in `slarre` occurred, If *info* = 2, internal error in `?larrv` occurred.

Application Notes

Currently `?stegr` is only set up to find *all* the *n* eigenvalues and eigenvectors of *T* in O(n^2) time, that is, only *range* ='A' is supported.

Currently the routine `?stein` is called when an appropriate σ_i cannot be chosen in step (c) above. `?stein` invokes modified Gram-Schmidt when eigenvalues are close.

`?stegr` works only on machines which follow IEEE-754 floating-point standard in their handling of infinities and NaNs. Normal execution of `?stegr` may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not conform to the IEEE-754 standard.

?pteqr

Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric positive-definite tridiagonal matrix.

Syntax

```
call spteqr ( compz, n, d, e, z, ldz, work, info )
call dpteqr ( compz, n, d, e, z, ldz, work, info )
call cpteqr ( compz, n, d, e, z, ldz, work, info )
call zpteqr ( compz, n, d, e, z, ldz, work, info )
```

Description

This routine computes all the eigenvalues and (optionally) all the eigenvectors of a real symmetric positive-definite tridiagonal matrix T. In other words, the routine can compute the spectral factorization: $T = Z\Lambda Z^T$.
Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i; Z is an orthogonal matrix whose columns are eigenvectors. Thus,

$$Tz_i = \lambda_i z_i \text{ for } i = 1, 2, \ldots, n.$$

(The routine normalizes the eigenvectors so that $\|z_i\|_2 = 1$.)

You can also use the routine for computing the eigenvalues and eigenvectors of real symmetric (or complex Hermitian) positive-definite matrices A reduced to tridiagonal form T: $A = QTQ^H$. In this case, the spectral factorization is as follows: $A = QTQ^H = (QZ)\Lambda(QZ)^H$. Before calling `?pteqr`, you must reduce A to tridiagonal form and generate the explicit matrix Q by calling the following routines:

	for real matrices:	for complex matrices:
full storage	`?sytrd`, `?orgtr`	`?hetrd`, `?ungtr`
packed storage	`?sptrd`, `?opgtr`	`?hptrd`, `?upgtr`
band storage	`?sbtrd` (*vect*='V')	`?hbtrd` (*vect*='V')

The routine first factorizes T as LDL^H where L is a unit lower bidiagonal matrix, and D is a diagonal matrix. Then it forms the bidiagonal matrix
$B = LD^{1/2}$ and calls `?bdsqr` to compute the singular values of B, which are the same as the eigenvalues of T.

Input Parameters

compz	`CHARACTER*1`. Must be `'N'` or `'I'` or `'V'`. If *compz* = `'N'`, the routine computes eigenvalues only. If *compz* = `'I'`, the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix *T*. If *compz* = `'V'`, the routine computes the eigenvalues and eigenvectors of *A* (and the array *z* must contain the matrix *Q* on entry).
n	`INTEGER`. The order of the matrix *T* ($n \geq 0$).
d, *e*, *work*	`REAL` for single-precision flavors `DOUBLE PRECISION` for double-precision flavors. Arrays: *d*(*) contains the diagonal elements of *T*. The dimension of *d* must be at least max(1, *n*). *e*(*) contains the off-diagonal elements of *T*. The dimension of *e* must be at least max(1, *n*-1). *work*(*) is a workspace array. The dimension of *work* must be: at least 1 if *compz* = `'N'`; at least max(1, 4**n*-4) if *compz* = `'V'` or `'I'`.
z	`REAL` for `spteqr` `DOUBLE PRECISION` for `dpteqr` `COMPLEX` for `cpteqr` `DOUBLE COMPLEX` for `zpteqr`. Array, `DIMENSION` (*ldz*, *) If *compz* = `'N'` or `'I'`, *z* need not be set. If *vect* = `'V'`, *z* must contains the *n* by *n* matrix *Q*. The second dimension of *z* must be: at least 1 if *compz* = `'N'`; at least max(1, *n*) if *compz* = `'V'` or `'I'`.
ldz	`INTEGER`. The first dimension of *z*. Constraints: $ldz \geq 1$ if *compz* = `'N'`; $ldz \geq \max(1, n)$ if *compz* = `'V'` or `'I'`.

Output Parameters

d	The *n* eigenvalues in descending order, unless *info* > 0. See also *info*.

`e` On exit, the array is overwritten.

`z` If `info` = 0, contains the `n` orthonormal eigenvectors, stored by columns. (The `i`th column corresponds to the `i`th eigenvalue.)

`info` `INTEGER`.
If `info` = 0, the execution is successful.
If `info` = `i`, the leading minor of order `i` (and hence *T* itself) is not positive-definite.
If `info` = `n` + `i`, the algorithm for computing singular values failed to converge; `i` off-diagonal elements have not converged to zero.
If `info` = `-i`, the `i`th parameter had an illegal value.

Application Notes

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then

$$|\mu_i - \lambda_i| \le c(n)\varepsilon K\lambda_i$$

where $c(n)$ is a modestly increasing function of n, ε is the machine precision, and $K = \|DTD\|_2 \|(DTD)^{-1}\|_2$, D is diagonal with $d_{ii} = t_{ii}^{-1/2}$.

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$$\theta(u_i, w_i) \le c(n)\varepsilon K / \min_{i\ne j}(|\lambda_i - \lambda_j|/|\lambda_i + \lambda_j|).$$

Here $\min_{i\ne j}(|\lambda_i - \lambda_j|/|\lambda_i + \lambda_j|)$ is the *relative gap* between λ_i and the other eigenvalues.

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about

$30n^2$ if `compz` = `'N'`;

$6n^3$ (for complex flavors, $12n^3$) if `compz` = `'V'` or `'I'`.

?stebz

Computes selected eigenvalues of a real symmetric tridiagonal matrix by bisection.

Syntax

```
call sstebz (range, order, n, vl, vu, il, iu, abstol,
   d, e, m, nsplit, w, iblock, isplit, work, iwork, info)
call dstebz (range, order, n, vl, vu, il, iu, abstol,
   d, e, m, nsplit, w, iblock, isplit, work, iwork, info)
```

Description

This routine computes some (or all) of the eigenvalues of a real symmetric tridiagonal matrix T by bisection. The routine searches for zero or negligible off-diagonal elements to see if T splits into block-diagonal form
$T = \mathrm{diag}(T_1, T_2, \ldots)$. Then it performs bisection on each of the blocks T_i and returns the block index of each computed eigenvalue, so that a subsequent call to `?stein` can also take advantage of the block structure.

Input Parameters

range — `CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`.
If *range* = `'A'`, the routine computes all eigenvalues.
If *range* = `'V'`, the routine computes eigenvalues λ_i in the half-open interval: $vl < \lambda_i \le vu$.
If *range* = `'I'`, the routine computes eigenvalues with indices *il* to *iu*.

order — `CHARACTER*1`. Must be `'B'` or `'E'`.
If *order* = `'B'`, the eigenvalues are to be ordered from smallest to largest within each split-off block.
If *order* = `'E'`, the eigenvalues for the entire matrix are to be ordered from smallest to largest.

n — `INTEGER`. The order of the matrix T ($n \ge 0$).

vl, vu — `REAL` for `sstebz`
`DOUBLE PRECISION` for `dstebz`.
If *range* = `'V'`, the routine computes eigenvalues λ_i in the half-open interval: $vl < \lambda_i \le vu$.

If *range* = `'A'` or `'I'`, *vl* and *vu* are not referenced.

il, iu — INTEGER. Constraint: $1 \le il \le iu \le n$.
If *range* = 'I', the routine computes eigenvalues λ_i such that $il \le i \le iu$ (assuming that the eigenvalues λ_i are in ascending order).

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol — REAL for sstebz
DOUBLE PRECISION for dstebz.
The absolute tolerance to which each eigenvalue is required. An eigenvalue (or cluster) is considered to have converged if it lies in an interval of width *abstol*. If *abstol* ≤ 0.0, then the tolerance is taken as $\varepsilon\|T\|_1$, where ε is the machine precision.

d, e — REAL for sstebz
DOUBLE PRECISION for dstebz.
Arrays:
d(*) contains the diagonal elements of *T*.
The dimension of *d* must be at least max(1, *n*).

e(*) contains the off-diagonal elements of *T*.
The dimension of *e* must be at least max(1, *n*-1).

iwork — INTEGER. Workspace.
Array, DIMENSION at least max(1, 3*n*).

Output Parameters

m — INTEGER. The actual number of eigenvalues found.

nsplit — INTEGER. The number of diagonal blocks detected in *T*.

w — REAL for sstebz
DOUBLE PRECISION for dstebz.
Array, DIMENSION at least max(1, *n*).
The computed eigenvalues, stored in *w*(1) to *w*(*m*).

iblock, isplit — INTEGER.
Arrays, DIMENSION at least max(1, *n*).
A positive value *iblock*(*i*) is the block number of the eigenvalue stored in *w*(*i*) (see also *info*).
The leading *nsplit* elements of *isplit* contain points at which *T* splits into blocks T_i as follows: the block
T_1 contains rows/columns 1 to *isplit*(1); the block
T_2 contains rows/columns *isplit*(1)+1 to *isplit*(2), and so on.

info `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = 1, for *range* = `'A'` or `'V'`, the algorithm failed to compute some of the required eigenvalues to the desired accuracy; *iblock*(*i*)< 0 indicates that the eigenvalue stored in *w*(*i*) failed to converge.
If *info* = 2, for *range* = `'I'`, the algorithm failed to compute some of the required eigenvalues. Try calling the routine again with *range* = `'A'`.
If *info* = 3:
 for *range* = `'A'` or `'V'`, same as *info* = 1;
 for *range* = `'I'`, same as *info* = 2.
If *info* = 4, no eigenvalues have been computed. The floating-point arithmetic on the computer is not behaving as expected.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The eigenvalues of *T* are computed to high relative accuracy which means that if they vary widely in magnitude, then any small eigenvalues will be computed more accurately than, for example, with the standard *QR* method. However, the reduction to tridiagonal form (prior to calling the routine) may exclude the possibility of obtaining high relative accuracy in the small eigenvalues of the original matrix if its eigenvalues vary widely in magnitude.

?stein

Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix.

Syntax

```
call sstein ( n, d, e, m, w, iblock, isplit, z, ldz,
   work, iwork, ifailv, info )
call dstein ( n, d, e, m, w, iblock, isplit, z, ldz,
   work, iwork, ifailv, info )
call cstein ( n, d, e, m, w, iblock, isplit, z, ldz,
   work, iwork, ifailv, info )
call zstein ( n, d, e, m, w, iblock, isplit, z, ldz,
   work, iwork, ifailv, info )
```

Description

This routine computes the eigenvectors of a real symmetric tridiagonal matrix *T* corresponding to specified eigenvalues, by inverse iteration. It is designed to be used in particular after the specified eigenvalues have been computed by `?stebz` with *order* = 'B', but may also be used when the eigenvalues have been computed by other routines. If you use this routine after `?stebz`, it can take advantage of the block structure by performing inverse iteration on each block T_i separately, which is more efficient than using the whole matrix *T*.

If *T* has been formed by reduction of a full symmetric or Hermitian matrix *A* to tridiagonal form, you can transform eigenvectors of *T* to eigenvectors of *A* by calling `?ormtr` or `?opmtr` (for real flavors) or by calling `?unmtr` or `?upmtr` (for complex flavors).

Input Parameters

n	`INTEGER`. The order of the matrix *T* ($n \geq 0$).
m	`INTEGER`. The number of eigenvectors to be returned.
d, e, w	`REAL` for single-precision flavors `DOUBLE PRECISION` for double-precision flavors. Arrays: *d*(*) contains the diagonal elements of *T*. The dimension of *d* must be at least max(1, *n*).

e(*) contains the off-diagonal elements of *T*.
The dimension of *e* must be at least max(1, *n*-1).

w(*) contains the eigenvalues of *T*, stored in *w*(1) to *w*(*m*) (as returned by ?stebz, see page 4-140). Eigenvalues of T_1 must be supplied first, in non-decreasing order; then those of T_2, again in non-decreasing order, and so on. Constraint:
if *iblock*(*i*) = *iblock*(*i*+1), *w*(*i*) ≤ *w*(*i*+1).

The dimension of *w* must be at least max(1, *n*).

iblock,*isplit* — INTEGER.
Arrays, DIMENSION at least max(1, *n*).
The arrays *iblock* and *isplit*, as returned by ?stebz with *order* = 'B'.

If you did not call ?stebz with *order* = 'B', set all elements of *iblock* to 1, and *isplit*(1) to *n*.)

ldz — INTEGER. The first dimension of the output array *z*; *ldz* ≥ max(1, *n*).

work — REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors. Workspace array, DIMENSION at least max(1, 5*n*).

iwork — INTEGER.
Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

z — REAL for sstein
DOUBLE PRECISION for dstein
COMPLEX for cstein
DOUBLE COMPLEX for zstein.
Array, DIMENSION (*ldz*, *).
If *info* = 0, *z* contains the *m* orthonormal eigenvectors, stored by columns. (The *i*th column corresponds to the *i*th specified eigenvalue.)

ifailv — INTEGER. Array, DIMENSION at least max(1, *m*).
If *info* = *i* > 0, the first *i* elements of *ifailv* contain the indices of any eigenvectors that failed to converge.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = *i*, then *i* eigenvectors (as indicated by the parameter *ifailv*) each

failed to converge in 5 iterations. The current iterates are stored in the corresponding columns of the array *z*.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

Each computed eigenvector z_i is an exact eigenvector of a matrix $T + E_i$, where $\|E_i\|_2 = O(\varepsilon)\|T\|_2$. However, a set of eigenvectors computed by this routine may not be orthogonal to so high a degree of accuracy as those computed by `?steqr`.

?disna

Computes the reciprocal condition numbers for the eigenvectors of a symmetric/ Hermitian matrix or for the left or right singular vectors of a general matrix.

Syntax

```
call sdisna (job, m, n, d, sep, info)
call ddisna (job, m, n, d, sep, info)
```

Description

This routine computes the reciprocal condition numbers for the eigenvectors of a real symmetric or complex Hermitian matrix or for the left or right singular vectors of a general *m*-by-*n* matrix.

The reciprocal condition number is the 'gap' between the corresponding eigenvalue or singular value and the nearest other one.

The bound on the error, measured by angle in radians, in the *i*-th computed vector is given by

`slamch`('E') * (*anorm* / *sep*(*i*))

where *anorm* = $\|A\|_2$ = max(|*d*(j)|). *sep*(*i*) is not allowed to be smaller than `slamch`('E')**anorm* in order to limit the size of the error bound.

`?disna` may also be used to compute error bounds for eigenvectors of the generalized symmetric definite eigenproblem.

Input Parameters

job — `CHARACTER*1`. Must be `'E'`,`'L'` , or `'R'`.
Specifies for which problem the reciprocal condition numbers should be computed:
job = `'E'`: for the eigenvectors of a symmetric/Hermitian matrix ;
job = `'L'`: for the left singular vectors of a general matrix;
job = `'R'`: for the right singular vectors of a general matrix .

m — `INTEGER`. The number of rows of the matrix ($m \geq 0$).

n — `INTEGER`. If *job* = `'L'`, or `'R'`, the number of columns of the matrix ($n \geq 0$). Ignored if *job* = `'E'`.

d — `REAL` for `sdisna`
`DOUBLE PRECISION` for `ddisna`.
Array, dimension at least max(1,*m*) if *job* = `'E'`, and at least max(1, min(*m*, *n*)) if *job* = `'L'`or `'R'`.
This array must contain the eigenvalues (if *job* = `'E'`) or singular values (if *job* = `'L'` or `'R'`) of the matrix, in either increasing or decreasing order. If singular values, they must be non-negative.

Output Parameters

sep — `REAL` for `sdisna`
`DOUBLE PRECISION` for `ddisna`.
Array, dimension at least max(1,*m*) if *job* = `'E'`, and at least max(1, min(*m*, *n*)) if *job* = `'L'`or `'R'`.
The reciprocal condition numbers of the vectors.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.

Generalized Symmetric-Definite Eigenvalue Problems

Generalized symmetric-definite eigenvalue problems are as follows: find the eigenvalues λ and the corresponding eigenvectors *z* that satisfy one of these equations:

$$Az = \lambda Bz, \quad ABz = \lambda z, \text{ or } BAz = \lambda z$$

where *A* is an *n* by *n* symmetric or Hermitian matrix, and *B* is an *n* by *n* symmetric positive-definite or Hermitian positive-definite matrix.

In these problems, there exist *n* real eigenvectors corresponding to real eigenvalues (even for complex Hermitian matrices *A* and *B*).

Routines described in this section allow you to reduce the above generalized problems to standard symmetric eigenvalue problem $Cy = \lambda y$,
which you can solve by calling LAPACK routines described earlier in this chapter (see page 4-95).

Different routines allow the matrices to be stored either conventionally or in packed storage. Prior to reduction, the positive-definite matrix *B* must first be factorized using either `?potrf` or `?pptrf`.
The reduction routine for the banded matrices *A* and *B* uses a split Cholesky factorization for which a specific routine `?pbstf` is provided. This refinement halves the amount of work required to form matrix *C*.

Table 4-4 Computational Routines for Reducing Generalized Eigenproblems to Standard Problems

Matrix type	Reduce to standard problems (full storage)	Reduce to standard problems (packed storage)	Reduce to standard problems (band matrices)	Factorize band matrix
real symmetric matrices	`?sygst`	`?spgst`	`?sbgst`	`?pbstf`
complex Hermitian matrices	`?hegst` /	`?hpgst`	`?hbgst`	`?pbstf`

?sygst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form.

Syntax

```
call ssygst ( itype, uplo, n, a, lda, b, ldb, info )
call dsygst ( itype, uplo, n, a, lda, b, ldb, info )
```

Description

This routine reduces real symmetric-definite generalized eigenproblems

$$Az = \lambda Bz, \quad ABz = \lambda z, \quad \text{or} \quad BAz = \lambda z$$

to the standard form $Cy = \lambda y$. Here A is a real symmetric matrix, and B is a real symmetric positive-definite matrix. Before calling this routine, call `?potrf` to compute the Cholesky factorization: $B = U^TU$ or $B = LL^T$
(see page 3-12).

Input Parameters

itype — `INTEGER`. Must be 1 or 2 or 3.
If *itype* = 1, the generalized eigenproblem is $Az = \lambda Bz$;
for *uplo* = `'U'`: $C = U^{-T}AU^{-1}$, $z = U^{-1}y$;
for *uplo* = `'L'`: $C = L^{-1}AL^{-T}$, $z = L^{-T}y$.
If *itype* = 2, the generalized eigenproblem is $ABz = \lambda z$;
for *uplo* = `'U'`: $C = UAU^T$, $z = U^{-1}y$;
for *uplo* = `'L'`: $C = L^TAL$, $z = L^{-T}y$.
If *itype* = 3, the generalized eigenproblem is $BAz = \lambda z$;
for *uplo* = `'U'`: $C = UAU^T$, $z = U^Ty$;
for *uplo* = `'L'`: $C = L^TAL$, $z = Ly$.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, the array *a* stores the upper triangle of A; you must supply B in the factored form $B = U^TU$.
If *uplo* = `'L'`, the array *a* stores the lower triangle of A; you must supply B in the factored form $B = LL^T$.

n — `INTEGER`. The order of the matrices A and B ($n \geq 0$).

a, b REAL for ssygst
DOUBLE PRECISION for dsygst.
Arrays:
a(*lda*,*) contains the upper or lower triangle of A.
The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the Cholesky-factored matrix B:
$B = U^T U$ or $B = LL^T$ (as returned by ?potrf).
The second dimension of *b* must be at least max(1, *n*).

lda INTEGER. The first dimension of *a*; at least max(1, *n*).

ldb INTEGER. The first dimension of *b*; at least max(1, *n*).

Output Parameters

a The upper or lower triangle of A is overwritten by the upper or lower triangle of C, as specified by the arguments *itype* and *uplo*.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

Forming the reduced matrix C is a stable procedure. However, it involves implicit multiplication by B^{-1} (if *itype* = 1) or B (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if B is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3.

?hegst

Reduces a complex Hermitian-definite generalized eigenvalue problem to the standard form.

Syntax

```
call chegst ( itype, uplo, n, a, lda, b, ldb, info )
call zhegst ( itype, uplo, n, a, lda, b, ldb, info )
```

Description

This routine reduces complex Hermitian-definite generalized eigenvalue problems

$$Az = \lambda Bz, \quad ABz = \lambda z, \quad \text{or} \quad BAz = \lambda z$$

to the standard form $Cy = \lambda y$. Here the matrix A is complex Hermitian, and B is complex Hermitian positive-definite. Before calling this routine, you must call `?potrf` to compute the Cholesky factorization: $B = U^H U$ or $B = LL^H$ (see page 3-12).

Input Parameters

itype — `INTEGER`. Must be 1 or 2 or 3.
If *itype* = 1, the generalized eigenproblem is $Az = \lambda Bz$;
for *uplo* = `'U'`: $C = U^{-H}AU^{-1}$, $z = U^{-1}y$;
for *uplo* = `'L'`: $C = L^{-1}AL^{-H}$, $z = L^{-H}y$.
If *itype* = 2, the generalized eigenproblem is $ABz = \lambda z$;
for *uplo* = `'U'`: $C = UAU^H$, $z = U^{-1}y$;
for *uplo* = `'L'`: $C = L^H AL$, $z = L^{-H}y$.
If *itype* = 3, the generalized eigenproblem is $BAz = \lambda z$;
for *uplo* = `'U'`: $C = UAU^H$, $z = U^H y$;
for *uplo* = `'L'`: $C = L^H AL$, $z = Ly$.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, the array *a* stores the upper triangle of A; you must supply B in the factored form $B = U^H U$.
If *uplo* = `'L'`, the array *a* stores the lower triangle of A; you must supply B in the factored form $B = LL^H$.

n — `INTEGER`. The order of the matrices A and B ($n \ge 0$).

a, b — `COMPLEX` for `chegst`
`DOUBLE COMPLEX` for `zhegst`.
Arrays:
`a(lda,*)` contains the upper or lower triangle of A.
The second dimension of `a` must be at least max(1, *n*).

`b(ldb,*)` contains the Cholesky-factored matrix B:
$B = U^H U$ or $B = LL^H$ (as returned by `?potrf`).
The second dimension of `b` must be at least max(1, *n*).

lda — `INTEGER`. The first dimension of `a`; at least max(1, *n*).

ldb — `INTEGER`. The first dimension of `b`; at least max(1, *n*).

Output Parameters

a — The upper or lower triangle of A is overwritten by the upper or lower triangle of C, as specified by the arguments *itype* and *uplo*.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

Forming the reduced matrix C is a stable procedure. However, it involves implicit multiplication by B^{-1} (if *itype* = 1) or B (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if B is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3.

?spgst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form using packed storage.

Syntax

```
call sspgst ( itype, uplo, n, ap, bp, info )
call dspgst ( itype, uplo, n, ap, bp, info )
```

Description

This routine reduces real symmetric-definite generalized eigenproblems

$$Az = \lambda Bz, \quad ABz = \lambda z, \text{ or } BAz = \lambda z$$

to the standard form $Cy = \lambda y$, using packed matrix storage. Here A is a real symmetric matrix, and B is a real symmetric positive-definite matrix. Before calling this routine, call `?pptrf` to compute the Cholesky factorization: $B = U^TU$ or $B = LL^T$ (see page 3-14).

Input Parameters

itype — `INTEGER`. Must be 1 or 2 or 3.
If *itype* = 1, the generalized eigenproblem is $Az = \lambda Bz$;
for *uplo* = 'U': $C = U^{-T}AU^{-1}$, $z = U^{-1}y$;
for *uplo* = 'L': $C = L^{-1}AL^{-T}$, $z = L^{-T}y$.
If *itype* = 2, the generalized eigenproblem is $ABz = \lambda z$;
for *uplo* = 'U': $C = UAU^T$, $z = U^{-1}y$;
for *uplo* = 'L': $C = L^TAL$, $z = L^{-T}y$.
If *itype* = 3, the generalized eigenproblem is $BAz = \lambda z$;
for *uplo* = 'U': $C = UAU^T$, $z = U^Ty$;
for *uplo* = 'L': $C = L^TAL$, $z = Ly$.

uplo — `CHARACTER*1`. Must be 'U' or 'L'.
If *uplo* = 'U', *ap* stores the packed upper triangle of A; you must supply B in the factored form $B = U^TU$.
If *uplo* = 'L', *ap* stores the packed lower triangle of A; you must supply B in the factored form $B = LL^T$.

n — `INTEGER`. The order of the matrices A and B ($n \geq 0$).

ap, bp REAL for sspgst
DOUBLE PRECISION for dspgst.
Arrays:
ap(*) contains the packed upper or lower triangle of *A*.
The dimension of *ap* must be at least max(1, *n**(*n*+1)/2).

bp(*) contains the packed Cholesky factor of *B*
(as returned by ?pptrf with the same *uplo* value).
The dimension of *bp* must be at least max(1, *n**(*n*+1)/2).

Output Parameters

ap The upper or lower triangle of *A* is overwritten by the upper or lower triangle of *C*, as specified by the arguments *itype* and *uplo*.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

Forming the reduced matrix *C* is a stable procedure. However, it involves implicit multiplication by B^{-1} (if *itype* = 1) or *B* (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3.

?hpgst

Reduces a complex Hermitian-definite generalized eigenvalue problem to the standard form using packed storage.

Syntax

```
call chpgst ( itype, uplo, n, ap, bp, info )
call zhpgst ( itype, uplo, n, ap, bp, info )
```

Description

This routine reduces real symmetric-definite generalized eigenproblems

$$Az = \lambda Bz,\quad ABz = \lambda z,\ \text{or}\ \ BAz = \lambda z$$

to the standard form $Cy = \lambda y$, using packed matrix storage. Here A is a real symmetric matrix, and B is a real symmetric positive-definite matrix. Before calling this routine, you must call `?pptrf` to compute the Cholesky factorization: $B = U^HU$ or $B = LL^H$ (see page 3-14).

Input Parameters

itype — `INTEGER`. Must be 1 or 2 or 3.
If *itype* = 1, the generalized eigenproblem is $Az = \lambda Bz$;
for *uplo* = `'U'`: $C = U^{-H}AU^{-1},\ z = U^{-1}y$;
for *uplo* = `'L'`: $C = L^{-1}AL^{-H},\ z = L^{-H}y$.
If *itype* = 2, the generalized eigenproblem is $ABz = \lambda z$;
for *uplo* = `'U'`: $C = UAU^H,\ z = U^{-1}y$;
for *uplo* = `'L'`: $C = L^HAL,\ z = L^{-H}y$.
If *itype* = 3, the generalized eigenproblem is $BAz = \lambda z$;
for *uplo* = `'U'`: $C = UAU^H,\ z = U^Hy$;
for *uplo* = `'L'`: $C = L^HAL,\ z = Ly$.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, *ap* stores the packed upper triangle of A; you must supply B in the factored form $B = U^HU$.
If *uplo* = `'L'`, *ap* stores the packed lower triangle of A; you must supply B in the factored form $B = LL^H$.

n — `INTEGER`. The order of the matrices A and B ($n \ge 0$).

ap, bp — COMPLEX for chpgst
DOUBLE COMPLEX for zhpgst.
Arrays:
ap(*) contains the packed upper or lower triangle of *A*.
The dimension of *a* must be at least max(1, *n**(*n*+1)/2).

bp(*) contains the packed Cholesky factor of *B*
(as returned by ?pptrf with the same *uplo* value).
The dimension of *b* must be at least max(1, *n**(*n*+1)/2).

Output Parameters

ap — The upper or lower triangle of *A* is overwritten by the upper or lower triangle of *C*, as specified by the arguments *itype* and *uplo*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

Forming the reduced matrix *C* is a stable procedure. However, it involves implicit multiplication by B^{-1} (if *itype* = 1) or *B* (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3.

?sbgst

Reduces a real symmetric-definite generalized eigenproblem for banded matrices to the standard form using the factorization performed by ?pbstf.

Syntax

```
call ssbgst ( vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx,
   work, info )
call dsbgst ( vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx,
   work, info )
```

Description

To reduce the real symmetric-definite generalized eigenproblem $Az = \lambda Bz$ to the standard form $Cy = \lambda y$, where *A, B* and *C* are banded, this routine must be preceded by a call to `spbstf/dpbstf`, which computes the split Cholesky factorization of the positive-definite matrix B: $B = S^T S$. The split Cholesky factorization, compared with the ordinary Cholesky factorization, allows the work to be approximately halved.

This routine overwrites *A* with $C = X^T AX$, where $X = S^{-1}Q$ and *Q* is an orthogonal matrix chosen (implicitly) to preserve the bandwidth of *A*.
The routine also has an option to allow the accumulation of *X*, and then, if *z* is an eigenvector of *C*, *Xz* is an eigenvector of the original system.

Input Parameters

vect	`CHARACTER*1`. Must be `'N'` or `'V'`. If *vect* = `'N'`, then matrix *X* is not returned; If *vect* = `'V'`, then matrix *X* is returned.
uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. If *uplo* = `'U'`, *ab* stores the upper triangular part of *A*. If *uplo* = `'L'`, *ab* stores the lower triangular part of *A*.
n	`INTEGER`. The order of the matrices *A* and *B* ($n \geq 0$).
ka	`INTEGER`. The number of super- or sub-diagonals in *A* ($ka \geq 0$).
kb	`INTEGER`. The number of super- or sub-diagonals in *B* ($ka \geq kb \geq 0$).
ab, bb, work	`REAL` for `ssbgst` `DOUBLE PRECISION` for `dsbgst` *ab* (*ldab*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format. The second dimension of the array *ab* must be at least max(1, *n*). *bb* (*ldbb*,*) is an array containing the banded split Cholesky factor of *B* as specified by *uplo*, *n* and *kb* and returned by `spbstf/dpbstf`. The second dimension of the array *bb* must be at least max(1, *n*). *work(*)* is a workspace array, `DIMENSION` at least max(1, 2**n*)
ldab	`INTEGER`. The first dimension of the array *ab*; must be at least *ka*+1.
ldbb	`INTEGER`. The first dimension of the array *bb*; must be at least *kb*+1.

ldx — The first dimension of the output array x. Constraints:
if *vect* = 'N' , then *ldx* ≥ 1;
if *vect* = 'V' , then *ldx* ≥ max(1, *n*).

Output Parameters

ab — On exit, this array is overwritten by the upper or lower triangle of *C* as specified by *uplo*.

x — REAL for ssbgst
DOUBLE PRECISION for dsbgst
Array.
If *vect* = 'V', then *x*(*ldx*,*) contains the *n* by *n* matrix $X = S^{-1}Q$.
If *vect* = 'N', then *x* is not referenced.
The second dimension of x must be:
at least max(1, *n*), if *vect* = 'V';
at least 1, if *vect* = 'N'.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

Forming the reduced matrix *C* involves implicit multiplication by B^{-1}. When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.
The total number of floating-point operations is approximately $6n^2 * kb$, when *vect* = 'N'. Additional $(3/2)n^3 * (kb/ka)$ operations are required when *vect* = 'V'. All these estimates assume that both *ka* and *kb* are much less than *n*.

?hbgst

Reduces a complex Hermitian-definite generalized eigenproblem for banded matrices to the standard form using the factorization performed by `?pbstf`.

Syntax

```
call chbgst ( vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx,
   work, rwork, info )
call zhbgst ( vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx,
   work, rwork, info )
```

Description

To reduce the complex Hermitian-definite generalized eigenproblem $Az = \lambda Bz$ to the standard form $Cy = \lambda y$, where *A, B* and *C* are banded, this routine must be preceded by a call to `cpbstf/zpbstf`, which computes the split Cholesky factorization of the positive-definite matrix *B*: $B = S^H S$. The split Cholesky factorization, compared with the ordinary Cholesky factorization, allows the work to be approximately halved.

This routine overwrites *A* with $C = X^H AX$, where $X = S^{-1}Q$ and *Q* is a unitary matrix chosen (implicitly) to preserve the bandwidth of *A*.
The routine also has an option to allow the accumulation of *X*, and then, if *z* is an eigenvector of *C*, *Xz* is an eigenvector of the original system.

Input Parameters

vect	`CHARACTER*1`. Must be `'N'` or `'V'`. If *vect* = `'N'`, then matrix *X* is not returned; If *vect* = `'V'`, then matrix *X* is returned.
uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. If *uplo* = `'U'`, *ab* stores the upper triangular part of *A*. If *uplo* = `'L'`, *ab* stores the lower triangular part of *A*.
n	`INTEGER`. The order of the matrices *A* and *B* ($n \ge 0$).
ka	`INTEGER`. The number of super- or sub-diagonals in *A* ($ka \ge 0$).
kb	`INTEGER`. The number of super- or sub-diagonals in *B* ($ka \ge kb \ge 0$).

ab,bb,work COMPLEX for chbgst
DOUBLE COMPLEX for zhbgst
ab (*ldab*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format. The second dimension of the array *ab* must be at least max(1, *n*).
bb (*ldbb*,*) is an array containing the banded split Cholesky factor of *B* as specified by *uplo*, *n* and *kb* and returned by cpbstf/zpbstf. The second dimension of the array *bb* must be at least max(1, *n*).
work(*) is a workspace array, DIMENSION at least max(1, *n*)

ldab INTEGER. The first dimension of the array *ab*; must be at least *ka*+1.

ldbb INTEGER. The first dimension of the array *bb*; must be at least *kb*+1.

ldx The first dimension of the output array x. Constraints:
if *vect* ='N' , then *ldx* ≥ 1;
if *vect* ='V' , then *ldx* ≥ max(1, *n*).

rwork REAL for chbgst
DOUBLE PRECISION for zhbgst
Workspace array, DIMENSION at least max(1, *n*)

Output Parameters

ab On exit, this array is overwritten by the upper or lower triangle of *C* as specified by *uplo*.

x COMPLEX for chbgst
DOUBLE COMPLEX for zhbgst
Array.
If *vect* ='V', then *x* (*ldx*,*) contains the *n* by *n* matrix $X = S^{-1}Q$.
If *vect* ='N', then *x* is not referenced.
The second dimension of x must be:
at least max(1, *n*), if *vect* ='V';
at least 1, if *vect* ='N'.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

Forming the reduced matrix *C* involves implicit multiplication by B^{-1}. When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

The total number of floating-point operations is approximately $20n^2$*`kb`, when `vect` ='N'. Additional $5n^3$*(`kb/ka`) operations are required when `vect` ='V'. All these estimates assume that both `ka` and `kb` are much less than `n`.

?pbstf

Computes a split Cholesky factorization of a real symmetric or complex Hermitian positive-definite banded matrix used in `?sbgst/?hbgst`.

Syntax

```
call spbstf ( uplo, n, kb, bb, ldbb, info )
call dpbstf ( uplo, n, kb, bb, ldbb, info )
call cpbstf ( uplo, n, kb, bb, ldbb, info )
call zpbstf ( uplo, n, kb, bb, ldbb, info )
```

Description

This routine computes a split Cholesky factorization of a real symmetric or complex Hermitian positive-definite band matrix B. It is to be used in conjunction with `?sbgst/?hbgst`.

The factorization has the form $B = S^T S$ (or $B = S^H S$ for complex flavors), where S is a band matrix of the same bandwidth as B and the following structure: S is upper triangular in the first (n+kb)/2 rows and lower triangular in the remaining rows.

Input Parameters

uplo — `CHARACTER*1`. Must be 'U' or 'L'.
If *uplo* = 'U', *bb* stores the upper triangular part of B.
If *uplo* = 'L', *bb* stores the lower triangular part of B.

n — `INTEGER`. The order of the matrix B ($n \geq 0$).

kb — `INTEGER`. The number of super- or sub-diagonals in B ($kb \geq 0$).

bb — `REAL` for `spbstf`
`DOUBLE PRECISION` for `dpbstf`
`COMPLEX` for `cpbstf`
`DOUBLE COMPLEX` for `zpbstf`.

`bb(ldbb,*)` is an array containing either upper or lower triangular part of the matrix B (as specified by *uplo*) in band storage format.
The second dimension of the array *bb* must be at least max(1, *n*).

ldbb — INTEGER. The first dimension of *bb*; must be at least *kb*+1.

Output Parameters

bb — On exit, this array is overwritten by the elements of the split Cholesky factor S.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = *i*, then the factorization could not be completed, because the updated element b_{ii} would be the square root of a negative number; hence the matrix B is not positive-definite.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The computed factor S is the exact factor of a perturbed matrix $B + E$, where

$$|E| \le c(kb+1)\varepsilon|S^H||S|, \quad |e_{ij}| \le c(kb+1)\varepsilon\sqrt{b_{ii}b_{jj}}$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

The total number of floating-point operations for real flavors is approximately $n(kb+1)^2$. The number of operations for complex flavors is 4 times greater. All these estimates assume that *kb* is much less than *n*.

After calling this routine, you can call ?sbgst/?hbgst to solve the generalized eigenproblem $Az = \lambda Bz$, where A and B are banded and B is positive-definite.

Nonsymmetric Eigenvalue Problems

This section describes LAPACK routines for solving nonsymmetric eigenvalue problems, computing the Schur factorization of general matrices, as well as performing a number of related computational tasks.

A *nonsymmetric eigenvalue problem* is as follows: given a nonsymmetric (or non-Hermitian) matrix A, find the *eigenvalues* λ and the corresponding *eigenvectors* z that satisfy the equation

$$Az = \lambda z \quad \text{(right eigenvectors } z\text{)}$$

or the equation

$$z^H A = \lambda z^H \quad \text{(left eigenvectors } z\text{).}$$

Nonsymmetric eigenvalue problems have the following properties:

- The number of eigenvectors may be less than the matrix order (but is not less than the number of *distinct eigenvalues* of A).
- Eigenvalues may be complex even for a real matrix A.
- If a real nonsymmetric matrix has a complex eigenvalue $a+bi$ corresponding to an eigenvector z, then $a-bi$ is also an eigenvalue.
  The eigenvalue $a-bi$ corresponds to the eigenvector whose elements are complex conjugate to the elements of z.

To solve a nonsymmetric eigenvalue problem with LAPACK, you usually need to reduce the matrix to the upper Hessenberg form and then solve the eigenvalue problem with the Hessenberg matrix obtained. Table 4-5 lists LAPACK routines for reducing the matrix to the upper Hessenberg form by an orthogonal (or unitary) similarity transformation $A = QHQ^H$ as well as routines for solving eigenvalue problems with Hessenberg matrices, forming the Schur factorization of such matrices and computing the corresponding condition numbers.

Decision tree in Figure 4-4 helps you choose the right routine or sequence of routines for an eigenvalue problem with a real nonsymmetric matrix.
If you need to solve an eigenvalue problem with a complex non-Hermitian matrix, use the decision tree shown in Figure 4-5.

Table 4-5 Computational Routines for Solving Nonsymmetric Eigenvalue Problems

Operation performed	Routines for real matrices	Routines for complex matrices
Reduce to Hessenberg form $A = QHQ^H$	?gehrd,	?gehrd
Generate the matrix Q	?orghr	?unghr
Apply the matrix Q	?ormhr	?unmhr
Balance matrix	?gebal	?gebal
Transform eigenvectors of balanced matrix to those of the original matrix	?gebak	?gebak
Find eigenvalues and Schur factorization (QR algorithm)	?hseqr	?hseqr
Find eigenvectors from Hessenberg form (inverse iteration)	?hsein	?hsein
Find eigenvectors from Schur factorization	?trevc	?trevc
Estimate sensitivities of eigenvalues and eigenvectors	?trsna	?trsna
Reorder Schur factorization	?trexc	?trexc
Reorder Schur factorization, find the invariant subspace and estimate sensitivities	?trsen	?trsen
Solves Sylvester's equation.	?trsyl	?trsyl

Figure 4-4 Decision Tree: Real Nonsymmetric Eigenvalue Problems

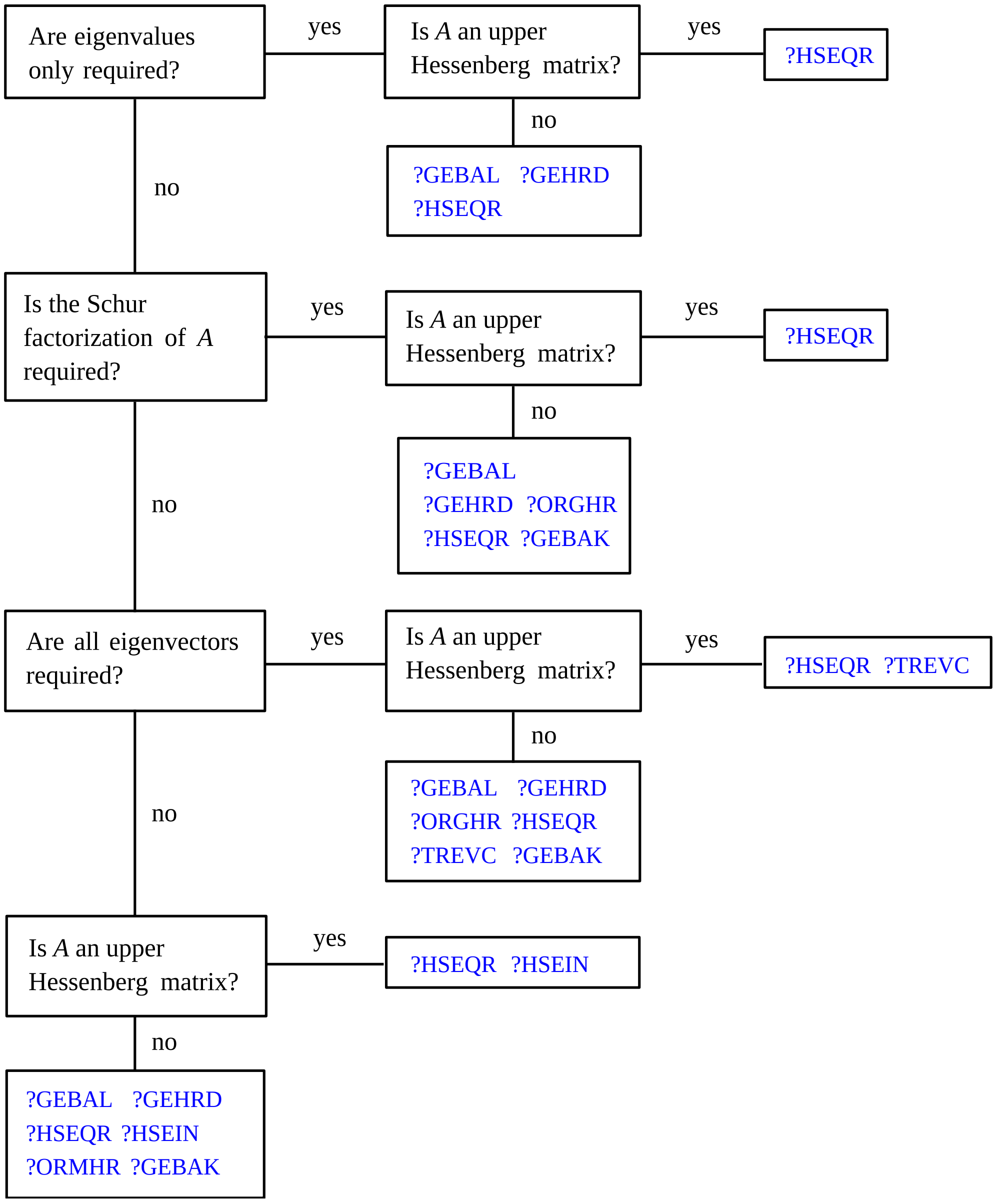
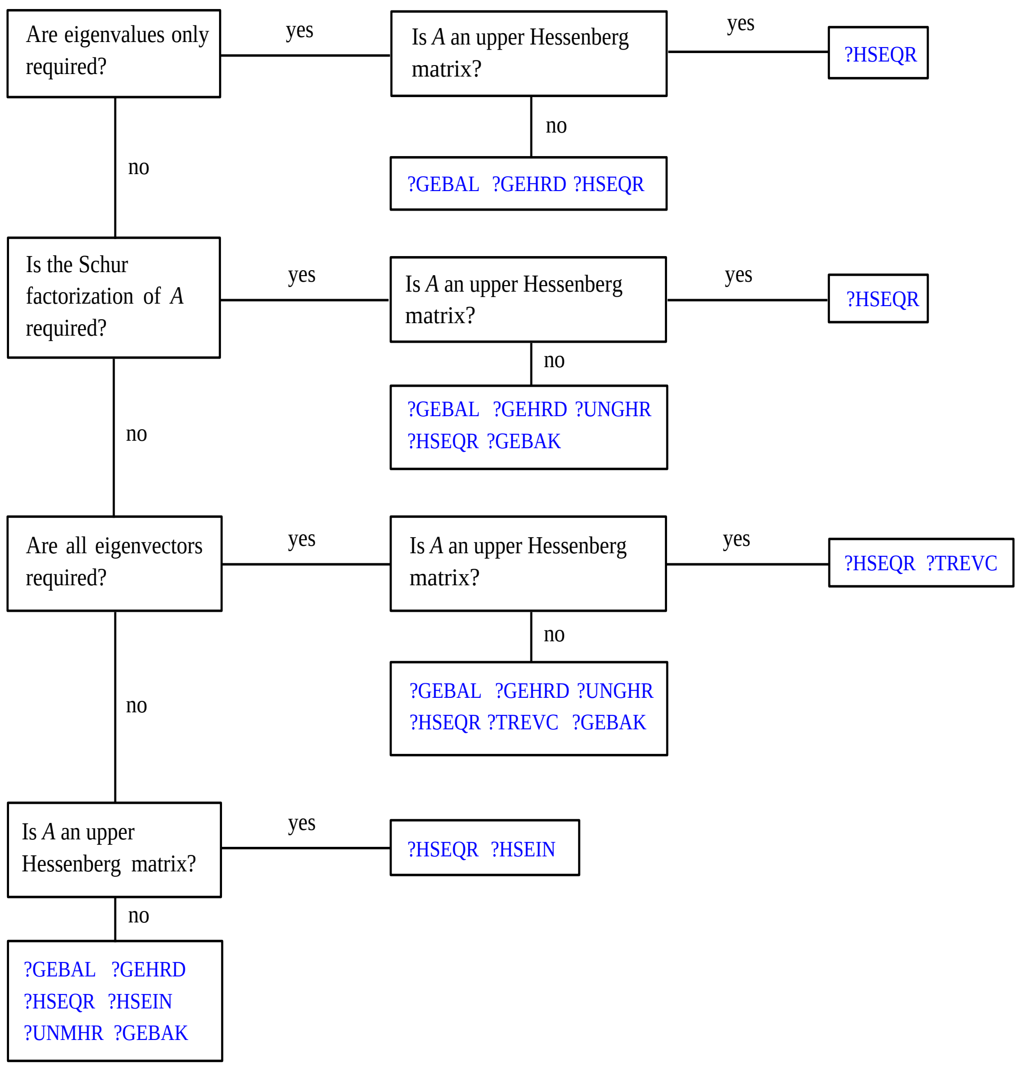


Figure 4-5 Decision Tree: Complex Non-Hermitian Eigenvalue Problems



?gehrd

Reduces a general matrix to upper Hessenberg form.

Syntax

```
call sgehrd ( n, ilo, ihi, a, lda, tau, work, lwork, info )
call dgehrd ( n, ilo, ihi, a, lda, tau, work, lwork, info )
call cgehrd ( n, ilo, ihi, a, lda, tau, work, lwork, info )
call zgehrd ( n, ilo, ihi, a, lda, tau, work, lwork, info )
```

Description

The routine reduces a general matrix A to upper Hessenberg form H by an orthogonal or unitary similarity transformation $A = QHQ^H$. Here H has real subdiagonal elements.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of *elementary reflectors*. Routines are provided to work with Q in this representation.

Input Parameters

n	`INTEGER`. The order of the matrix A ($n \ge 0$).
ilo, ihi	`INTEGER`. If A has been output by `?gebal`, then *ilo* and *ihi* must contain the values returned by that routine. Otherwise *ilo* = 1 and *ihi* = *n*. (If $n > 0$, then $1 \le ilo \le ihi \le n$; if $n = 0$, *ilo* = 1 and *ihi* = 0.)
a, work	`REAL` for `sgehrd` `DOUBLE PRECISION` for `dgehrd` `COMPLEX` for `cgehrd` `DOUBLE COMPLEX` for `zgehrd`. Arrays: *a* (*lda*, *) contains the matrix A. The second dimension of *a* must be at least max(1, *n*). *work* (*lwork*) is a workspace array.
lda	`INTEGER`. The first dimension of *a*; at least max(1, *n*).
lwork	`INTEGER`. The size of the *work* array; at least max(1,*n*). See *Application notes* for the suggested value of *lwork*.

Output Parameters

a — Overwritten by the upper Hessenberg matrix *H* and details of the matrix *Q*. The subdiagonal elements of *H* are real.

tau — REAL for sgehrd
DOUBLE PRECISION for dgehrd
COMPLEX for cgehrd
DOUBLE COMPLEX for zgehrd.
Array, DIMENSION at least max (1, *n*-1).
Contains additional information on the matrix *Q*.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* = *n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The computed Hessenberg matrix *H* is exactly similar to a nearby matrix $A + E$, where $||E||_2 < c(n)\varepsilon||A||_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision.

The approximate number of floating-point operations for real flavors is $(2/3)(ihi - ilo)^2(2ihi + 2ilo + 3n)$; for complex flavors it is 4 times greater.

?orghr

Generates the real orthogonal matrix Q determined by ?gehrd.

Syntax

```
call sorghr ( n, ilo, ihi, a, lda, tau, work, lwork, info )
call dorghr ( n, ilo, ihi, a, lda, tau, work, lwork, info )
```

Description

This routine explicitly generates the orthogonal matrix Q that has been determined by a preceding call to sgehrd/dgehrd. (The routine ?gehrd reduces a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation, $A = QHQ^T$, and represents the matrix Q as a product of *ihi*-*ilo* *elementary reflectors*. Here *ilo* and *ihi* are values determined by sgebal/dgebal when balancing the matrix; if the matrix has not been balanced, *ilo* = 1 and *ihi* = *n*.)

The matrix Q generated by ?orghr has the structure:

$$Q = \begin{bmatrix} I & 0 & 0 \\ 0 & Q_{22} & 0 \\ 0 & 0 & I \end{bmatrix}$$

where Q_{22} occupies rows and columns *ilo* to *ihi*.

Input Parameters

n	INTEGER. The order of the matrix Q ($n \geq 0$).
ilo, ihi	INTEGER. These must be the same parameters *ilo* and *ihi*, respectively, as supplied to ?gehrd. (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, $ilo = 1$ and $ihi = 0$.)
a, tau, work	REAL for sorghr DOUBLE PRECISION for dorghr Arrays: *a*(*lda*,*) contains details of the vectors which define the elementary reflectors, as returned by ?gehrd. The second dimension of *a* must be at least max(1, *n*).

tau(*) contains further details of the elementary reflectors, as returned by ?gehrd.
The dimension of *tau* must be at least max (1, *n*-1).

work(*lwork*) is a workspace array.

lda — INTEGER. The first dimension of *a*; at least max(1, *n*).

lwork — INTEGER. The size of the *work* array;
lwork ≥ max(1,*ihi*–*ilo*).
See *Application notes* for the suggested value of *lwork*.

Output Parameters

a — Overwritten by the *n* by *n* orthogonal matrix *Q*.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* =(*ihi*–*ilo*)**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The computed matrix *Q* differs from the exact result by a matrix *E* such that $||E||_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is $(4/3)(ihi-ilo)^3$.

The complex counterpart of this routine is ?unghr.

?ormhr

Multiplies an arbitrary real matrix C by the real orthogonal matrix Q determined by `?gehrd`.

Syntax

```
call sormhr ( side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc,
              work, lwork, info )
call dormhr ( side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc,
              work, lwork, info )
```

Description

This routine multiplies a matrix C by the orthogonal matrix Q that has been determined by a preceding call to `sgehrd/dgehrd`. (The routine `?gehrd` reduces a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation, $A = QHQ^T$, and represents the matrix Q as a product of *ihi*-*ilo* *elementary reflectors*. Here *ilo* and *ihi* are values determined by `sgebal/dgebal` when balancing the matrix; if the matrix has not been balanced, *ilo* = 1 and *ihi* = *n*.)

With `?ormhr`, you can form one of the matrix products QC, Q^TC, CQ, or CQ^T, overwriting the result on C (which may be any real rectangular matrix).

A common application of `?ormhr` is to transform a matrix V of eigenvectors of H to the matrix QV of eigenvectors of A.

Input Parameters

side	`CHARACTER*1`. Must be `'L'` or `'R'`. If *side* = `'L'`, then the routine forms QC or Q^TC. If *side* = `'R'`, then the routine forms CQ or CQ^T.
trans	`CHARACTER*1`. Must be `'N'` or `'T'`. If *trans* = `'N'`, then Q is applied to C. If *trans* = `'T'`, then Q^T is applied to C.
m	`INTEGER`. The number of rows in C ($m \geq 0$).
n	`INTEGER`. The number of columns in C ($n \geq 0$).

ilo, ihi INTEGER. These must be the same parameters *ilo* and *ihi*, respectively, as supplied to ?gehrd.
If $m > 0$ and *side* ='L', then $1 \le ilo \le ihi \le m$.
If $m = 0$ and *side* ='L', then *ilo* = 1 and *ihi* = 0.
If $n > 0$ and *side* ='R', then $1 \le ilo \le ihi \le n$.
If $n = 0$ and *side* ='R', then *ilo* = 1 and *ihi* = 0.

a,tau,c,work REAL for sormhr
DOUBLE PRECISION for dormhr
Arrays:
a(*lda*,*) contains details of the vectors which define the *elementary reflectors*, as returned by ?gehrd.
The second dimension of *a* must be at least max(1, *m*) if *side* = 'L' and at least max(1, *n*) if *side* = 'R'.

tau(*) contains further details of the *elementary reflectors*, as returned by ?gehrd.
The dimension of *tau* must be at least max (1, *m*-1) if *side* ='L' and at least max (1, *n*-1) if *side* ='R'.

c(*ldc*,*) contains the *m* by *n* matrix *C*.
The second dimension of *c* must be at least max(1, *n*).

work (*lwork*) is a workspace array.

lda INTEGER. The first dimension of *a*; at least max(1, *m*) if *side* ='L' and at least max (1, *n*) if *side* ='R'.

ldc INTEGER. The first dimension of *c*; at least max(1, *m*).

lwork INTEGER. The size of the *work* array.
If *side* ='L', *lwork* ≥ max(1,*n*).
If *side* ='R', *lwork* ≥ max(1,*m*).
See *Application notes* for the suggested value of *lwork*.

Output Parameters

c *C* is overwritten by *QC* or Q^TC or CQ^T or *CQ* as specified by *side* and *trans*.

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, `lwork` should be at least `n*`*blocksize* if `side` = `'L'` and at least `m*`*blocksize* if `side` = `'R'`, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The computed matrix Q differs from the exact result by a matrix E such that $||E||_2 = O(\varepsilon)||C||_2$, where ε is the machine precision.

The approximate number of floating-point operations is
$2n(ihi-ilo)^2$ if `side` = `'L'`;
$2m(ihi-ilo)^2$ if `side` = `'R'`.

The complex counterpart of this routine is `?unmhr`.

?unghr

Generates the complex unitary matrix Q determined by `?gehrd`.

Syntax

```
call cunghr ( n, ilo, ihi, a, lda, tau, work, lwork, info )
call zunghr ( n, ilo, ihi, a, lda, tau, work, lwork, info )
```

Description

This routine is intended to be used following a call to `cgehrd`/`zgehrd`, which reduces a complex matrix A to upper Hessenberg form H by a unitary similarity transformation: $A = QHQ^H$. `?gehrd` represents the matrix Q as a product of *ihi*−*ilo* *elementary reflectors*. Here *ilo* and *ihi* are values determined by `cgebal`/`zgebal` when balancing the matrix; if the matrix has not been balanced, *ilo* = 1 and *ihi* = *n*.

Use the routine `?unghr` to generate Q explicitly as a square matrix. The matrix Q has the structure:

$$Q = \begin{bmatrix} I & 0 & 0 \\ 0 & Q_{22} & 0 \\ 0 & 0 & I \end{bmatrix}$$

where Q_{22} occupies rows and columns *ilo* to *ihi*.

Input Parameters

n — `INTEGER`. The order of the matrix Q ($n \ge 0$).

ilo, ihi — `INTEGER`. These must be the same parameters *ilo* and *ihi*, respectively, as supplied to `?gehrd`. (If $n > 0$, then $1 \le ilo \le ihi \le n$. If $n = 0$, then $ilo = 1$ and $ihi = 0$.)

a, tau, work — `COMPLEX` for `cunghr`
`DOUBLE COMPLEX` for `zunghr`.
Arrays:
`a(lda,*)` contains details of the vectors which define the *elementary reflectors*, as returned by `?gehrd`.
The second dimension of *a* must be at least max(1, *n*).

	`tau(*)` contains further details of the *elementary reflectors*, as returned by `?gehrd`. The dimension of `tau` must be at least max (1, `n-1`).
	`work (lwork)` is a workspace array.
`lda`	`INTEGER`. The first dimension of `a`; at least max(1, `n`).
`lwork`	`INTEGER`. The size of the `work` array; `lwork` ≥ max(1, `ihi–ilo`). See *Application notes* for the suggested value of `lwork`.

Output Parameters

`a`	Overwritten by the `n` by `n` unitary matrix *Q*.
`work(1)`	If `info` = 0, on exit `work(1)` contains the minimum value of `lwork` required for optimum performance. Use this `lwork` for subsequent runs.
`info`	`INTEGER`. If `info` = 0, the execution is successful. If `info` = `-i`, the `i`th parameter had an illegal value.

Application Notes

For better performance, try using `lwork` = (`ihi–ilo`)**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The computed matrix *Q* differs from the exact result by a matrix *E* such that $||E||_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of real floating-point operations is $(16/3)(ihi-ilo)^3$.

The real counterpart of this routine is `?orghr`.

?unmhr

Multiplies an arbitrary complex matrix C by the complex unitary matrix Q determined by `?gehrd`.

Syntax

```
call cunmhr ( side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc,
              work, lwork, info )
call zunmhr ( side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc,
              work, lwork, info )
```

Description

This routine multiplies a matrix C by the unitary matrix Q that has been determined by a preceding call to `cgehrd`/`zgehrd`. (The routine `?gehrd` reduces a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation, $A = QHQ^H$, and represents the matrix Q as a product of *ihi*-*ilo* *elementary reflectors*. Here *ilo* and *ihi* are values determined by `cgebal`/`zgebal` when balancing the matrix; if the matrix has not been balanced, *ilo* = 1 and *ihi* = *n*.)

With `?unmhr`, you can form one of the matrix products QC, Q^HC, CQ, or CQ^H, overwriting the result on C (which may be any complex rectangular matrix). A common application of this routine is to transform a matrix V of eigenvectors of H to the matrix QV of eigenvectors of A.

Input Parameters

side	`CHARACTER*1`. Must be `'L'` or `'R'`. If *side* = `'L'`, then the routine forms QC or Q^HC. If *side* = `'R'`, then the routine forms CQ or CQ^H.
trans	`CHARACTER*1`. Must be `'N'` or `'C'`. If *trans* = `'N'`, then Q is applied to C. If *trans* = `'T'`, then Q^H is applied to C.
m	`INTEGER`. The number of rows in C ($m \ge 0$).
n	`INTEGER`. The number of columns in C ($n \ge 0$).
ilo, ihi	`INTEGER`. These must be the same parameters *ilo* and *ihi*, respectively, as supplied to `?gehrd`. If $m > 0$ and *side* =`'L'`, then $1 \le ilo \le ihi \le m$.

If m = 0 and *side* = 'L', then *ilo* = 1 and *ihi* = 0.
If $n > 0$ and *side* = 'R', then $1 \le ilo \le ihi \le n$.
If n = 0 and *side* = 'R', then *ilo* =1 and *ihi* = 0.

a, tau, c, work — COMPLEX for cunmhr
DOUBLE COMPLEX for zunmhr.
Arrays:
a (*lda*,*) contains details of the vectors which define the elementary reflectors, as returned by ?gehrd.
The second dimension of *a* must be at least max(1, *m*) if *side* = 'L' and at least max(1, *n*) if *side* = 'R'.

tau(*) contains further details of the elementary reflectors, as returned by ?gehrd.
The dimension of *tau* must be at least max (1, *m*-1) if *side* = 'L' and at least max (1, *n*-1) if *side* = 'R'.

c (*ldc*,*) contains the *m* by *n* matrix *C*.
The second dimension of *c* must be at least max(1, *n*).

work (*lwork*) is a workspace array.

lda — INTEGER. The first dimension of *a*; at least max(1, *m*) if *side* = 'L' and at least max (1, *n*) if *side* = 'R'.

ldc — INTEGER. The first dimension of *c*; at least max(1, *m*).

lwork — INTEGER. The size of the *work* array.
If *side* = 'L', $lwork \ge \max(1,n)$.
If *side* = 'R', $lwork \ge \max(1,m)$.
See *Application notes* for the suggested value of *lwork*.

Output Parameters

c — *C* is overwritten by QC or $Q^H C$ or CQ^H or CQ as specified by *side* and *trans*.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, *lwork* should be at least *n***blocksize* if *side* = 'L' and at least *m***blocksize* if *side* = 'R', where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The computed matrix Q differs from the exact result by a matrix E such that $||E||_2 = O(\varepsilon)\ ||C||_2$, where ε is the machine precision.

The approximate number of floating-point operations is
$8n(ihi-ilo)^2$ if *side* = 'L';
$8m(ihi-ilo)^2$ if *side* = 'R'.

The real counterpart of this routine is ?ormhr.

?gebal

Balances a general matrix to improve the accuracy of computed eigenvalues and eigenvectors.

Syntax

```
call sgebal ( job, n, a, lda, ilo, ihi, scale, info )
call dgebal ( job, n, a, lda, ilo, ihi, scale, info )
call cgebal ( job, n, a, lda, ilo, ihi, scale, info )
call zgebal ( job, n, a, lda, ilo, ihi, scale, info )
```

Description

This routine *balances* a matrix *A* by performing either or both of the following two similarity transformations:

(1) The routine first attempts to permute *A* to block upper triangular form:

$$PAP^T = A' = \begin{bmatrix} A'_{11} & A'_{12} & A'_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & 0 & A'_{33} \end{bmatrix}$$

where *P* is a permutation matrix, and A'_{11} and A'_{33} are upper triangular. The diagonal elements of A'_{11} and A'_{33} are eigenvalues of *A*. The rest of the eigenvalues of *A* are the eigenvalues of the central diagonal block A'_{22}, in rows and columns `ilo` to `ihi`. Subsequent operations to compute the eigenvalues of *A* (or its Schur factorization) need only be applied to these rows and columns; this can save a significant amount of work if $ilo > 1$ and $ihi < n$. If no suitable permutation exists (as is often the case), the routine sets $ilo = 1$ and $ihi = n$, and A'_{22} is the whole of *A*.

(2) The routine applies a diagonal similarity transformation to A', to make the rows and columns of A'_{22} as close in norm as possible:

$$A'' = DA'D^{-1} = \begin{bmatrix} I & 0 & 0 \\ 0 & D_{22} & 0 \\ 0 & 0 & I \end{bmatrix} \times \begin{bmatrix} A'_{11} & A'_{12} & A'_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & 0 & A'_{33} \end{bmatrix} \times \begin{bmatrix} I & 0 & 0 \\ 0 & D_{22}^{-1} & 0 \\ 0 & 0 & I \end{bmatrix}$$

This scaling can reduce the norm of the matrix (that is, $\|A''_{22}\| < \|A'_{22}\|$), and hence reduce the effect of rounding errors on the accuracy of computed eigenvalues and eigenvectors.

Input Parameters

job — CHARACTER*1. Must be 'N' or 'P' or 'S' or 'B'.
If *job* ='N', then A is neither permuted nor scaled (but *ilo*, *ihi*, and *scale* get their values).
If *job* ='P', then A is permuted but not scaled.
If *job* ='S', then A is scaled but not permuted.
If *job* ='B', then A is both scaled and permuted.

n — INTEGER. The order of the matrix A ($n \geq 0$).

a — REAL for sgebal
DOUBLE PRECISION for dgebal
COMPLEX for cgebal
DOUBLE COMPLEX for zgebal.
Arrays:
a (*lda*,*) contains the matrix A.
The second dimension of *a* must be at least max(1, *n*).
a is not referenced if *job* ='N'.

lda — INTEGER. The first dimension of *a*; at least max(1, *n*).

Output Parameters

a — Overwritten by the balanced matrix (*a* is not referenced if *job* = 'N').

ilo, ihi — INTEGER. The values *ilo* and *ihi* such that on exit *a*(*i*,*j*) is zero if $i > j$ and $1 \leq j <$ *ilo* or *ihi* $< i \leq n$. If *job* ='N' or 'S', then *ilo* = 1 and *ihi* = *n*.

scale — REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors
Array, DIMENSION at least max(1, *n*).

Contains details of the permutations and scaling factors.

More precisely, if p_j is the index of the row and column interchanged with row and column j, and d_j is the scaling factor used to balance row and column j, then
scale(j)= p_j for j = 1, 2, . . . , *ilo*-1, *ihi*+1, . . . , *n*;
scale(j)= d_j for j = *ilo*, *ilo* + 1, . . . , *ihi*.
The order in which the interchanges are made is
n to *ihi*+1, then 1 to *ilo*-1.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The errors are negligible, compared with those in subsequent computations.

If the matrix *A* is balanced by this routine, then any eigenvectors computed subsequently are eigenvectors of the matrix *A*″ and hence you must call `?gebak` (see page 4-181) to transform them back to eigenvectors of *A*.

If the Schur vectors of *A* are required, do not call this routine with *job* = `'S'` or `'B'`, because then the balancing transformation is not orthogonal (not unitary for complex flavors). If you call this routine with *job* = `'P'`, then any Schur vectors computed subsequently are Schur vectors of the matrix *A*″, and you'll need to call `?gebak` (with *side* =`'R'`) to transform them back to Schur vectors of *A*.

The total number of floating-point operations is proportional to n^2.

?gebak

Transforms eigenvectors of a balanced matrix to those of the original nonsymmetric matrix.

Syntax

```
call sgebak ( job,side,n,ilo,ihi,scale,m,v,ldv,info )
call dgebak ( job,side,n,ilo,ihi,scale,m,v,ldv,info )
call cgebak ( job,side,n,ilo,ihi,scale,m,v,ldv,info )
call zgebak ( job,side,n,ilo,ihi,scale,m,v,ldv,info )
```

Description

This routine is intended to be used after a matrix A has been balanced by a call to `?gebal`, and eigenvectors of the balanced matrix A''_{22} have subsequently been computed. For a description of balancing, see `?gebal` (page 4-178). The balanced matrix A'' is obtained as $A'' = DPAP^TD^{-1}$, where P is a permutation matrix and D is a diagonal scaling matrix. This routine transforms the eigenvectors as follows:
if x is a right eigenvector of A'', then $P^TD^{-1}x$ is a right eigenvector of A;
if x is a left eigenvector of A'', then P^TDy is a left eigenvector of A.

Input Parameters

job — `CHARACTER*1`. Must be `'N'` or `'P'` or `'S'` or `'B'`.
The same parameter *job* as supplied to `?gebal`.

side — `CHARACTER*1`. Must be `'L'` or `'R'`.
If *side* = `'L'`, then left eigenvectors are transformed.
If *side* = `'R'`, then right eigenvectors are transformed.

n — `INTEGER`. The number of rows of the matrix of eigenvectors ($n \ge 0$).

ilo, ihi — `INTEGER`. The values *ilo* and *ihi*, as returned by `?gebal`. (If $n > 0$, then $1 \le ilo \le ihi \le n$;
if $n = 0$, then $ilo = 1$ and $ihi = 0$.)

scale — `REAL` for single-precision flavors
`DOUBLE PRECISION` for double-precision flavors
Array, `DIMENSION` at least max(1, *n*).

Contains details of the permutations and/or the scaling factors used to balance the original general matrix, as returned by `?gebal`.

m — `INTEGER`. The number of columns of the matrix of eigenvectors ($m \geq 0$).

v — `REAL` for `sgebak`
`DOUBLE PRECISION` for `dgebak`
`COMPLEX` for `cgebak`
`DOUBLE COMPLEX` for `zgebak`.
Arrays:
v(*ldv*,*) contains the matrix of left or right eigenvectors to be transformed.
The second dimension of *v* must be at least max(1, *m*).

ldv — `INTEGER`. The first dimension of *v*; at least max(1, *n*).

Output Parameters

v — Overwritten by the transformed eigenvectors.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The errors in this routine are negligible.

The approximate number of floating-point operations is approximately proportional to *m***n*.

?hseqr

Computes all eigenvalues and (optionally) the Schur factorization of a matrix reduced to Hessenberg form.

Syntax

```
call shseqr (job,compz,n,ilo,ihi,h,ldh,wr,wi,z,ldz,work,lwork,info)
call dhseqr (job,compz,n,ilo,ihi,h,ldh,wr,wi,z,ldz,work,lwork,info)
call chseqr (job,compz,n,ilo,ihi,h,ldh,w,z,ldz,work,lwork,info)
call zhseqr (job,compz,n,ilo,ihi,h,ldh,w,z,ldz,work,lwork,info)
```

Description

This routine computes all the eigenvalues, and optionally the Schur factorization, of an upper Hessenberg matrix H: $H = ZTZ^H$, where T is an upper triangular (or, for real flavors, quasi-triangular) matrix (the Schur form of H), and Z is the unitary or orthogonal matrix whose columns are the Schur vectors z_i.

You can also use this routine to compute the Schur factorization of a general matrix A which has been reduced to upper Hessenberg form H:
$A = QHQ^H$, where Q is unitary (orthogonal for real flavors);
$A = (QZ)T(QZ)^H$.

In this case, after reducing A to Hessenberg form by `?gehrd` (page 4-166), call `?orghr` to form Q explicitly (page 4-168) and then pass Q to `?hseqr` with *compz* = `'V'`.

You can also call `?gebal` (page 4-178) to balance the original matrix before reducing it to Hessenberg form by `?hseqr`, so that the Hessenberg matrix H will have the structure:

$$\begin{bmatrix} H_{11} & H_{12} & H_{13} \\ 0 & H_{22} & H_{23} \\ 0 & 0 & H_{33} \end{bmatrix}$$

where H_{11} and H_{33} are upper triangular.

If so, only the central diagonal block H_{22} (in rows and columns `ilo` to `ihi`) needs to be further reduced to Schur form (the blocks H_{12} and H_{23} are also affected). Therefore the values of `ilo` and `ihi` can be supplied to `?hseqr` directly. Also, after calling this routine you must call `?gebak` (page 4-181) to permute the Schur vectors of the balanced matrix to those of the original matrix.

If `?gebal` has not been called, however, then `ilo` must be set to 1 and `ihi` to *n*. Note that if the Schur factorization of *A* is required, `?gebal` must not be called with `job` = `'S'` or `'B'`, because the balancing transformation is not unitary (for real flavors, it is not orthogonal).

`?hseqr` uses a multishift form of the upper Hessenberg *QR* algorithm. The Schur vectors are normalized so that $||z_i||_2 = 1$, but are determined only to within a complex factor of absolute value 1 (for the real flavors, to within a factor ±1).

Input Parameters

job — `CHARACTER*1`. Must be `'E'` or `'S'`.
If *job* = `'E'`, then eigenvalues only are required.
If *job* = `'S'`, then the Schur form *T* is required.

compz — `CHARACTER*1`. Must be `'N'` or `'I'` or `'V'`.
If *compz* = `'N'`, then no Schur vectors are computed (and the array *z* is not referenced).
If *compz* = `'I'`, then the Schur vectors of *H* are computed (and the array *z* is initialized by the routine).
If *compz* = `'V'`, then the Schur vectors of *A* are computed (and the array *z* must contain the matrix *Q* on entry).

n — `INTEGER`. The order of the matrix *H* ($n \geq 0$).

ilo, ihi — `INTEGER`. If *A* has been balanced by `?gebal`, then *ilo* and *ihi* must contain the values returned by `?gebal`. Otherwise, *ilo* must be set to 1 and *ihi* to *n*.

h, z, work — `REAL` for `shseqr`
`DOUBLE PRECISION` for `dhseqr`
`COMPLEX` for `chseqr`
`DOUBLE COMPLEX` for `zhseqr`.
Arrays:
h(*ldh*,*) The *n* by *n* upper Hessenberg matrix *H*.
The second dimension of *h* must be at least max(1, *n*).

z(*ldz*,*)
If *compz* = `'V'`, then *z* must contain the matrix *Q* from the reduction to Hessenberg form.
If *compz* = `'I'`, then *z* need not be set.

If *compz* = 'N', then *z* is not referenced.
The second dimension of *z* must be
at least max(1, *n*) if *compz* = 'V' or 'I';
at least 1 if *compz* = 'N'.

work(*lwork*) is a workspace array.
The dimension of *work* must be at least max (1, *n*).

ldh INTEGER. The first dimension of *h*; at least max(1, *n*).

ldz INTEGER. The first dimension of *z*;
If *compz* = 'N', then *ldz* ≥ 1.
If *compz* = 'V' or 'I', then *ldz* ≥ max(1,*n*).

lwork INTEGER. The dimension of the array *work*.
lwork ≥ max(1,*n*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

w COMPLEX for chseqr
DOUBLE COMPLEX for zhseqr.
Array, DIMENSION at least max (1, *n*).
Contains the computed eigenvalues, unless *info*>0. The eigenvalues are stored in the same order as on the diagonal of the Schur form *T* (if computed).

wr, wi REAL for shseqr
DOUBLE PRECISION for dhseqr
Arrays, DIMENSION at least max (1, *n*) each.
Contain the real and imaginary parts, respectively, of the computed eigenvalues, unless *info* > 0. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first. The eigenvalues are stored in the same order as on the diagonal of the Schur form *T* (if computed).

z If *compz* = 'V' or 'I', then *z* contains the unitary (orthogonal) matrix of the required Schur vectors, unless *info* > 0.
If *compz* = 'N', then *z* is not referenced.

work(1) On exit, if *info* = 0, then *work(1)* returns the optimal *lwork*.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.
If *info* > 0, the algorithm has failed to find all the eigenvalues after a total 30(*ihi*–*ilo*+1) iterations. If *info* = *i*, elements 1,2, ..., *ilo*–1 and *i*+1, *i*+2, ..., *n* of *wr* and *wi* contain the real and imaginary parts of the eigenvalues which have been found.

Application Notes

The computed Schur factorization is the exact factorization of a nearby matrix $H + E$, where $\|E\|_2 < O(\varepsilon) \|H\|_2/s_i$, and ε is the machine precision.
If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then $|\lambda_i - \mu_i| \le c(n)\varepsilon \|H\|_2/s_i$ where $c(n)$ is a modestly increasing function of n, and s_i is the reciprocal condition number of λ_i. You can compute the condition numbers s_i by calling `?trsna` (see page 4-196).

The total number of floating-point operations depends on how rapidly the algorithm converges; typical numbers are as follows.

If only eigenvalues are computed:	$7n^3$ for real flavors $25n^3$ for complex flavors.
If the Schur form is computed:	$10n^3$ for real flavors $35n^3$ for complex flavors.
If the full Schur factorization is computed:	$20n^3$ for real flavors $70n^3$ for complex flavors.

?hsein

Computes selected eigenvectors of an upper Hessenberg matrix that correspond to specified eigenvalues.

Syntax

```
call shsein ( job, eigsrc, initv, select, n, h, ldh, wr, wi, vl,
    ldvl, vr, ldvr, mm, m, work, ifaill, ifailr, info )
call dhsein ( job, eigsrc, initv, select, n, h, ldh, wr, wi, vl,
    ldvl, vr, ldvr, mm, m, work, ifaill, ifailr, info )
call chsein ( job, eigsrc, initv, select, n, h, ldh, w, vl,
    ldvl, vr, ldvr, mm, m, work, rwork, ifaill, ifailr, info )
call zhsein ( job, eigsrc, initv, select, n, h, ldh, w, vl,
    ldvl, vr, ldvr, mm, m, work, rwork, ifaill, ifailr, info )
```

Description

This routine computes left and/or right eigenvectors of an upper Hessenberg matrix H, corresponding to selected eigenvalues.

The right eigenvector x and the left eigenvector y, corresponding to an eigenvalue λ, are defined by: $Hx = \lambda x$ and $y^H H = \lambda y^H$ (or $H^H y = \lambda^* y$).
Here λ^* denotes the conjugate of λ.

The eigenvectors are computed by inverse iteration. They are scaled so that, for a real eigenvector x, $\max|x_i| = 1$, and for a complex eigenvector, $\max(|\mathrm{Re}x_i| + |\mathrm{Im}x_i|) = 1$.

If H has been formed by reduction of a general matrix A to upper Hessenberg form, then eigenvectors of H may be transformed to eigenvectors of A by `?ormhr` (page 4-170) or `?unmhr` (page 4-175).

Input Parameters

job — `CHARACTER*1`. Must be `'R'` or `'L'` or `'B'`.
If *job* = `'R'`, then only right eigenvectors are computed.
If *job* = `'L'`, then only left eigenvectors are computed.
If *job* = `'B'`, then all eigenvectors are computed.

eigsrc	`CHARACTER*1`. Must be `'Q'` or `'N'`. If *eigsrc* = `'Q'`, then the eigenvalues of *H* were found using `?hseqr` (see page 4-183); thus if *H* has any zero sub-diagonal elements (and so is block triangular), then the *j*th eigenvalue can be assumed to be an eigenvalue of the block containing the *j*th row/column. This property allows the routine to perform inverse iteration on just one diagonal block. If *eigsrc* = `'N'`, then no such assumption is made and the routine performs inverse iteration using the whole matrix.
initv	`CHARACTER*1`. Must be `'N'` or `'U'`. If *initv* = `'N'`, then no initial estimates for the selected eigenvectors are supplied. If *initv* = `'U'`, then initial estimates for the selected eigenvectors are supplied in *vl* and/or *vr*.
select	`LOGICAL`. Array, `DIMENSION` at least max (1, *n*). Specifies which eigenvectors are to be computed. *For real flavors*: To obtain the real eigenvector corresponding to the real eigenvalue *wr*(*j*), set *select*(*j*) to `.TRUE.` To select the complex eigenvector corresponding to the complex eigenvalue (*wr*(*j*),*wi*(*j*)) with complex conjugate (*wr*(*j*+1),*wi*(*j*+1)), set *select*(*j*) and/or *select*(*j*+1) to `.TRUE.`; the eigenvector corresponding to the first eigenvalue in the pair is computed. *For complex flavors:* To select the eigenvector corresponding to the eigenvalue *w*(*j*), set *select*(*j*) to `.TRUE.`
n	`INTEGER`. The order of the matrix *H* ($n \geq 0$).
h,*vl*,*vr*,*work*	`REAL` for `shsein` `DOUBLE PRECISION` for `dhsein` `COMPLEX` for `chsein` `DOUBLE COMPLEX` for `zhsein`. Arrays: *h*(*ldh*,*) The *n* by *n* upper Hessenberg matrix *H*. The second dimension of *h* must be at least max(1, *n*). *vl*(*ldvl*,*) If *initv* = `'V'` and *job* = `'L'` or `'B'`, then *vl* must contain starting vectors for inverse iteration for the left eigenvectors. Each starting vector must be stored in the same column or columns as will be used to store the corresponding

eigenvector.
If *initv* = 'N', then *vl* need not be set.
The second dimension of *vl* must be at least max(1, *mm*) if *job* = 'L' or 'B' and at least 1 if *job* = 'R'.
The array *vl* is not referenced if *job* = 'R'.

vr(*ldvr*,*)
If *initv* = 'V' and *job* = 'R' or 'B', then *vr* must contain starting vectors for inverse iteration for the right eigenvectors. Each starting vector must be stored in the same column or columns as will be used to store the corresponding eigenvector.
If *initv* = 'N', then *vr* need not be set.
The second dimension of *vr* must be at least max(1, *mm*) if *job* = 'R' or 'B' and at least 1 if *job* = 'L'.
The array *vr* is not referenced if *job* = 'L'.

work(*) is a workspace array.
DIMENSION at least max (1, *n**(*n*+2)) for real flavors and at least max (1, *n***n*) for complex flavors.

ldh — INTEGER. The first dimension of *h*; at least max(1, *n*).

w — COMPLEX for chsein
DOUBLE COMPLEX for zhsein.
Array, DIMENSION at least max (1, *n*).
Contains the eigenvalues of the matrix *H*.
If *eigsrc* = 'Q', the array must be exactly as returned by ?hseqr.

wr, wi — REAL for shsein
DOUBLE PRECISION for dhsein
Arrays, DIMENSION at least max (1, *n*) each.
Contain the real and imaginary parts, respectively, of the eigenvalues of the matrix *H*. Complex conjugate pairs of values must be stored in consecutive elements of the arrays. If *eigsrc* = 'Q', the arrays must be exactly as returned by ?hseqr.

ldvl — INTEGER. The first dimension of *vl*.
If *job* = 'L' or 'B', *ldvl* ≥ max(1,*n*).
If *job* = 'R', *ldvl* ≥ 1.

ldvr — INTEGER. The first dimension of *vr*.
If *job* = 'R' or 'B', *ldvr* ≥ max(1,*n*).
If *job* = 'L', *ldvr* ≥ 1.

mm — `INTEGER`. The number of columns in *vl* and/or *vr*.
Must be at least *m*, the actual number of columns required (see *Output Parameters* below).
For real flavors, *m* is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector (see *select*).
For complex flavors, m is the number of selected eigenvectors (see *select*).
Constraint: $0 \le mm \le n$.

rwork — `REAL` for `chsein`
`DOUBLE PRECISION` for `zhsein`.
Array, `DIMENSION` at least max (1, *n*).

Output Parameters

select — Overwritten for real flavors only. If a complex eigenvector was selected as specified above, then *select*(*j*) is set to `.TRUE.` and *select*(*j*+1) to `.FALSE.`

w — The real parts of some elements of *w* may be modified, as close eigenvalues are perturbed slightly in searching for independent eigenvectors.

wr — Some elements of *wr* may be modified, as close eigenvalues are perturbed slightly in searching for independent eigenvectors.

vl, vr — If *job* = 'L' or 'B', *vl* contains the computed left eigenvectors (as specified by *select*).
If *job* = 'R' or 'B', *vr* contains the computed right eigenvectors (as specified by *select*).

The eigenvectors are stored consecutively in the columns of the array, in the same order as their eigenvalues.
For real flavors: a real eigenvector corresponding to a selected real eigenvalue occupies one column;
a complex eigenvector corresponding to a selected complex eigenvalue occupies two columns: the first column holds the real part and the second column holds the imaginary part.

m — `INTEGER`. *For real flavors*: the number of columns of *vl* and/or *vr* required to store the selected eigenvectors.
For complex flavors: the number of selected eigenvectors.

ifaill,ifailr — `INTEGER`.
Arrays, `DIMENSION` at least max(1, *mm*) each.
ifaill(*i*) = 0 if the *i*th column of *vl* converged;
ifaill(*i*) = *j* > 0 if the eigenvector stored in the *i*th column of *vl*

(corresponding to the *j*th eigenvalue) failed to converge.
ifailr(*i*) = 0 if the *i*th column of *vr* converged;
ifailr(*i*) = *j* > 0 if the eigenvector stored in the *i*th column of *vr* (corresponding to the *j*th eigenvalue) failed to converge.
For real flavors: if the *i*th and (*i*+1)th columns of *vl* contain a selected complex eigenvector, then *ifaill*(*i*) and *ifaill*(*i*+1) are set to the same value. A similar rule holds for *vr* and *ifailr*.

The array *ifaill* is not referenced if *job* ='R'.
The array *ifailr* is not referenced if *job* ='L'.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* > 0, then *i* eigenvectors (as indicated by the parameters *ifaill* and/or *ifailr* above) failed to converge. The corresponding columns of *vl* and/or *vr* contain no useful information.

Application Notes

Each computed right eigenvector x_i is the exact eigenvector of a nearby matrix $A + E_i$, such that $||E_i|| < O(\varepsilon)||A||$. Hence the residual is small:
$||Ax_i - \lambda_i x_i|| = O(\varepsilon)||A||$.

However, eigenvectors corresponding to close or coincident eigenvalues may not accurately span the relevant subspaces.

Similar remarks apply to computed left eigenvectors.

?trevc

Computes selected eigenvectors of an upper (quasi-) triangular matrix computed by ?hseqr.

Syntax

```
call strevc ( side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr,
   mm, m, work, info )
call dtrevc ( side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr,
   mm, m, work, info )
call ctrevc ( side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr,
   mm, m, work, rwork, info )
call ztrevc ( side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr,
   mm, m, work, rwork, info )
```

Description

This routine computes some or all of the right and/or left eigenvectors of an upper triangular matrix T (or, for real flavors, an upper quasi-triangular matrix T). Matrices of this type are produced by the Schur factorization of a general matrix: $A = QTQ^H$, as computed by ?hseqr (see page 4-183).

The right eigenvector x and the left eigenvector y of T corresponding to an eigenvalue w, are defined by:

$$Tx = wx \quad , \qquad y^H T = w y^H$$

where y^H denotes the conjugate transpose of y.

The eigenvalues are not input to this routine, but are read directly from the diagonal blocks of T.

This routine returns the matrices X and/or Y of right and left eigenvectors of T, or the products QX and/or QY, where Q is an input matrix.
If Q is the orthogonal/unitary factor that reduces a matrix A to Schur form T, then QX and QY are the matrices of right and left eigenvectors of A.

Input Parameters

side — CHARACTER*1. Must be 'R' or 'L' or 'B'.
If *side* ='R', then only right eigenvectors are computed.
If *side* ='L', then only left eigenvectors are computed.
If *side* ='B', then all eigenvectors are computed.

howmny — CHARACTER*1. Must be 'A' or 'B' or 'S'.
If *howmny* ='A', then all eigenvectors (as specified by *side*) are computed.
If *howmny* ='B', then all eigenvectors (as specified by *side*) are computed and backtransformed by the matrices supplied in *vl* and *vr* .
If *howmny* ='S', then selected eigenvectors (as specified by *side* and *select*) are computed.

select — LOGICAL.
Array, DIMENSION at least max (1, *n*).
If *howmny*='S', *select* specifies which eigenvectors are to be computed.
If *howmny*= 'A'or 'B', *select* is not referenced.
For real flavors:
If ω_j is a real eigenvalue, the corresponding real eigenvector is computed if *select*(*j*) is .TRUE..
If ω_j and ω_{j+1} are the real and imaginary parts of a complex eigenvalue, the corresponding complex eigenvector is computed if either *select*(*j*) or *select*(*j*+1) is .TRUE. , and on exit *select*(*j*) is set to .TRUE.and *select*(*j*+1) is set to .FALSE..
For complex flavors:
The eigenvector corresponding to the *j*-th eigenvalue is computed if *select*(*j*) is .TRUE..

n — INTEGER. The order of the matrix *T* ($n \geq 0$).

t,vl,vr,work — REAL for strevc
DOUBLE PRECISION for dtrevc
COMPLEX for ctrevc
DOUBLE COMPLEX for ztrevc.
Arrays:
t(*ldt*,*) contains the *n* by *n* matrix *T* in Schur canonical form.
The second dimension of *t* must be at least max(1, *n*).

vl(*ldvl*,*)
If *howmny* ='B' and *side* ='L' or 'B', then *vl* must contain an n by n matrix *Q* (usually the matrix of Schur vectors returned by ?hseqr).
If *howmny* ='A' or 'S', then *vl* need not be set.
The second dimension of *vl* must be at least max(1, *mm*) if *side* ='L' or 'B' and at least 1 if *side* ='R'.
The array *vl* is not referenced if *side* ='R'.

vr (*ldvr*,*)
If *howmny* ='B' and *side* ='R' or 'B', then *vr* must contain an n by n matrix *Q* (usually the matrix of Schur vectors returned by ?hseqr). .

If *howmny* = 'A' or 'S', then *vr* need not be set.
The second dimension of *vr* must be at least max(1, *mm*) if *side* = 'R' or 'B' and at least 1 if *side* = 'L'.
The array *vr* is not referenced if *side* = 'L'.

work(*) is a workspace array.
DIMENSION at least max (1, 3*n) for real flavors and at least max (1, 2*n) for complex flavors.

ldt — INTEGER. The first dimension of *t*; at least max(1, *n*).

ldvl — INTEGER. The first dimension of *vl*.
If *side* = 'L' or 'B', *ldvl* ≥ max(1,*n*).
If *side* = 'R', *ldvl* ≥ 1.

ldvr — INTEGER. The first dimension of *vr*.
If *side* = 'R' or 'B', *ldvr* ≥ max(1,*n*).
If *side* = 'L', *ldvr* ≥ 1.

mm — INTEGER. The number of columns in the arrays *vl* and/or *vr*. Must be at least *m* (the precise number of columns required). If *howmny* = 'A' or 'B', $m = n$.
If *howmny* = 'S': *for real flavors*, *m* is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector;

for complex flavors, *m* is the number of selected eigenvectors (see *select*).
Constraint: $0 \le m \le n$.

rwork — REAL for ctrevc
DOUBLE PRECISION for ztrevc.
Workspace array, DIMENSION at least max (1, *n*).

Output Parameters

select — If a complex eigenvector of a real matrix was selected as specified above, then *select*(*j*) is set to .TRUE. and *select*(*j*+1) to .FALSE.

vl, *vr* — If *side* = 'L' or 'B', *vl* contains the computed left eigenvectors (as specified by *howmny* and *select*).
If *side* = 'R' or 'B', *vr* contains the computed right eigenvectors (as specified by *howmny* and *select*).

The eigenvectors are stored consecutively in the columns of the array, in the same order as their eigenvalues.
For real flavors: corresponding to each real eigenvalue is a real eigenvector,

occupying one column; corresponding to each complex conjugate pair of eigenvalues is a complex eigenvector, occupying two columns; the first column holds the real part and the second column holds the imaginary part.

m — `INTEGER`.
For complex flavors: the number of selected eigenvectors. If *howmny* = `'A'` or `'B'`, *m* is set to *n*.
For real flavors: the number of columns of *vl* and/or *vr* actually used to store the selected eigenvectors.
If *howmny* = `'A'` or `'B'`, *m* is set to *n*.

info — `INTEGER`. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

If x_i is an exact right eigenvector and y_i is the corresponding computed eigenvector, then the angle $\theta(y_i,x_i)$ between them is bounded as follows: $\theta(y_i,x_i) \le (c(n)\varepsilon\|T\|_2)/\text{sep}_i$ where sep_i is the reciprocal condition number
of x_i. The condition number sep_i may be computed by calling `?trsna`.

?trsna

Estimates condition numbers for specified eigenvalues and right eigenvectors of an upper (quasi-) triangular matrix.

Syntax

```
call strsna ( job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr,
   s, sep, mm, m, work, ldwork, iwork, info )
call dtrsna ( job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr,
   s, sep, mm, m, work, ldwork, iwork, info )
call ctrsna ( job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr,
   s, sep, mm, m, work, ldwork, rwork, info )
call ztrsna ( job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr,
   s, sep, mm, m, work, ldwork, rwork, info )
```

Description

This routine estimates condition numbers for specified eigenvalues and/or right eigenvectors of an upper triangular matrix T (or, for real flavors, upper quasi-triangular matrix T in canonical Schur form). These are the same as the condition numbers of the eigenvalues and right eigenvectors of an original matrix $A = ZTZ^H$ (with unitary or, for real flavors, orthogonal Z), from which T may have been derived.

The routine computes the reciprocal of the condition number of an eigenvalue λ_i as $s_i = |v^H u|/(||u||_E||v||_E)$, where u and v are the right and left eigenvectors of T, respectively, corresponding to λ_i. This reciprocal condition number always lies between zero (ill-conditioned) and one (well-conditioned).

An approximate error estimate for a computed eigenvalue λ_i is then given by $\varepsilon||T||/s_i$, where ε is the *machine precision*.

To estimate the reciprocal of the condition number of the right eigenvector corresponding to λ_i, the routine first calls `?trexc` (see page 4-201) to reorder the eigenvalues so that λ_i is in the leading position:

$$T = Q\begin{bmatrix}\lambda_i & C^H \\ 0 & T_{22}\end{bmatrix}Q^H$$

The reciprocal condition number of the eigenvector is then estimated as sep_i, the smallest singular value of the matrix $T_{22} - \lambda_i I$. This number ranges from zero (ill-conditioned) to very large (well-conditioned).

An approximate error estimate for a computed right eigenvector u corresponding to λ_i is then given by $\varepsilon \|T\|/sep_i$.

Input Parameters

job — `CHARACTER*1`. Must be `'E'` or `'V'` or `'B'`.
If *job* = `'E'`, then condition numbers for eigenvalues only are computed.
If *job* = `'V'`, then condition numbers for eigenvectors only are computed.
If *job* = `'B'`, then condition numbers for both eigenvalues and eigenvectors are computed.

howmny — `CHARACTER*1`. Must be `'A'` or `'S'`.
If *howmny* = `'A'`, then the condition numbers for all eigenpairs are computed.
If *howmny* = `'S'`, then condition numbers for selected eigenpairs (as specified by *select*) are computed.

select — `LOGICAL`.
Array, `DIMENSION` at least max (1, *n*) if *howmny* = `'S'` and at least 1 otherwise.
Specifies the eigenpairs for which condition numbers are to be computed if *howmny*= `'S'`.
For real flavors:
To select condition numbers for the eigenpair corresponding to the real eigenvalue λ_j, *select*(j) must be set `.TRUE.`; to select condition numbers for the eigenpair corresponding to a complex conjugate pair of eigenvalues λ_j and λ_{j+1}, *select*(j) and/or *select*(j+1) must be set `.TRUE.`
For complex flavors:
To select condition numbers for the eigenpair corresponding to the eigenvalue λ_j, *select*(j) must be set `.TRUE.`
select is not referenced if *howmny* = `'A'`.

n — `INTEGER`. The order of the matrix T ($n \ge 0$).

t, vl, vr, work — `REAL` for `strsna`
`DOUBLE PRECISION` for `dtrsna`
`COMPLEX` for `ctrsna`
`DOUBLE COMPLEX` for `ztrsna`.
Arrays:
t(*ldt*,*) contains the *n* by *n* matrix T.
The second dimension of *t* must be at least max(1, *n*).

vl(*ldvl*,*)
If *job* ='E' or 'B', then *vl* must contain the left eigenvectors of *T* (or of any matrix QTQ^H with *Q* unitary or orthogonal) corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vl*, as returned by ?trevc or ?hsein.
The second dimension of *vl* must be at least max(1, *mm*) if *job* ='E' or 'B' and at least 1 if *job* ='V'.
The array *vl* is not referenced if *job* ='V'.

vr(*ldvr*,*)
If *job* ='E' or 'B', then *vr* must contain the right eigenvectors of *T* (or of any matrix QTQ^H with *Q* unitary or orthogonal) corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vr*, as returned by ?trevc or ?hsein.
The second dimension of *vr* must be at least max(1, *mm*) if *job* ='E' or 'B' and at least 1 if *job* ='V'.
The array *vr* is not referenced if *job* ='V'.

work(*ldwork*,*) is a workspace array.
The second dimension of *work* must be
at least max(1, *n*+1) for complex flavors and
at least max(1, *n*+6) for real flavors if *job* ='V' or 'B'; at least 1 if *job* ='E'.
The array *work* is not referenced if *job* ='E'.

ldt INTEGER. The first dimension of *t*; at least max(1, *n*).

ldvl INTEGER. The first dimension of *vl*.
If *job* ='E' or 'B', *ldvl*≥max(1,*n*).
If *job* ='V', *ldvl*≥1.

ldvr INTEGER. The first dimension of *vr*.
If *job* ='E' or'B', *ldvr*≥max(1,*n*).
If *job* ='R', *ldvr*≥1.

mm INTEGER. The number of elements in the arrays *s* and *sep*, and the number of columns in *vl* and *vr* (if used). Must be at least *m* (the precise number required).
If *howmny* ='A', *m* = *n*;
if *howmny* ='S', *for real flavors* *m* is obtained by counting 1 for each selected real eigenvalue and 2 for each selected complex conjugate pair of eigenvalues.
for complex flavors *m* is the number of selected eigenpairs (see *select*).
Constraint: $0 \le m \le n$.

ldwork	INTEGER. The first dimension of *work*. If *job* = 'V' or 'B', *ldwork* ≥ max(1,*n*). If *job* = 'E', *ldwork* ≥ 1.
rwork	REAL for ctrsna, ztrsna. Array, DIMENSION at least max (1, *n*).
iwork	INTEGER for strsna, dtrsna. Array, DIMENSION at least max (1, *n*).

Output Parameters

s	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least max(1, *mm*) if *job* = 'E' or 'B' and at least 1 if *job* = 'V'. Contains the reciprocal condition numbers of the selected eigenvalues if *job* = 'E' or 'B', stored in consecutive elements of the array. Thus *s*(*j*), *sep*(*j*) and the *j*th columns of *vl* and *vr* all correspond to the same eigenpair (but not in general the *j*th eigenpair unless all eigenpairs have been selected). *For real flavors*: For a complex conjugate pair of eigenvalues, two consecutive elements of S are set to the same value. The array *s* is not referenced if *job* = 'V'.
sep	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least max(1, *mm*) if *job* = 'V' or 'B' and at least 1 if *job* = 'E'. Contains the estimated reciprocal condition numbers of the selected right eigenvectors if *job* = 'V' or 'B', stored in consecutive elements of the array. *For real flavors*: for a complex eigenvector, two consecutive elements of *sep* are set to the same value; if the eigenvalues cannot be reordered to compute *sep*(*j*), then *sep*(*j*) is set to zero; this can only occur when the true value would be very small anyway. The array *sep* is not referenced if *job* = 'E'.
m	INTEGER. *For complex flavors*: the number of selected eigenpairs. If *howmny* = 'A', *m* is set to *n*. *For real flavors*: the number of elements of s and/or *sep* actually used to store the estimated condition numbers. If *howmny* = 'A', *m* is set to *n*.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The computed values sep_i may overestimate the true value, but seldom by a factor of more than 3.

?trexc

Reorders the Schur factorization of a general matrix.

Syntax

```
call strexc ( compq, n, t, ldt, q, ldq, ifst, ilst, work, info )
call dtrexc ( compq, n, t, ldt, q, ldq, ifst, ilst, work, info )
call ctrexc ( compq, n, t, ldt, q, ldq, ifst, ilst, info )
call ztrexc ( compq, n, t, ldt, q, ldq, ifst, ilst, info )
```

Description

This routine reorders the Schur factorization of a general matrix $A = QTQ^H$, so that the diagonal element or block of T with row index `ifst` is moved to row `ilst`.

The reordered Schur form S is computed by an unitary (or, for real flavors, orthogonal) similarity transformation: $S = Z^H TZ$. Optionally the updated matrix P of Schur vectors is computed as $P = QZ$, giving $A = PSP^H$.

Input Parameters

compq	`CHARACTER*1`. Must be `'V'` or `'N'`. If *compq* = `'V'`, then the Schur vectors (Q) are updated. If *compq* = `'N'`, then no Schur vectors are updated.
n	`INTEGER`. The order of the matrix T ($n \geq 0$).
t, q	`REAL` for `strexc` `DOUBLE PRECISION` for `dtrexc` `COMPLEX` for `ctrexc` `DOUBLE COMPLEX` for `ztrexc`. Arrays: `t(ldt,*)` contains the *n* by *n* matrix *T*. The second dimension of `t` must be at least max(1, *n*). *q*(*ldq*,*) If *compq* = `'V'`, then *q* must contain *Q* (Schur vectors). If *compq* = `'N'`, then *q* is not referenced. The second dimension of *q* must be at least max(1, *n*) if *compq* = `'V'` and at least 1 if *compq* = `'N'`.

ldt — `INTEGER`. The first dimension of *t*; at least max(1, *n*).

ldq — `INTEGER`. The first dimension of *q*;
If *compq* = `'N'`, then *ldq*≥1.
If *compq* = `'V'`, then *ldq*≥max(1,*n*).

ifst, ilst — `INTEGER`. $1 \le ifst \le n$; $1 \le ilst \le n$.
Must specify the reordering of the diagonal elements (or blocks, which is possible for real flavors) of the matrix *T*. The element (or block) with row index *ifst* is moved to row *ilst* by a sequence of exchanges between adjacent elements (or blocks).

work — `REAL` for `strexc`
`DOUBLE PRECISION` for `dtrexc`.
Array, `DIMENSION` at least max (1, *n*).

Output Parameters

t — Overwritten by the updated matrix *S*.

q — If *compq* = `'V'`, *q* contains the updated matrix of Schur vectors.

ifst, ilst — Overwritten for real flavors only.
If *ifst* pointed to the second row of a 2 by 2 block on entry, it is changed to point to the first row; *ilst* always points to the first row of the block in its final position (which may differ from its input value by ±1).

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

The computed matrix *S* is exactly similar to a matrix $T + E$, where
$\|E\|_2 = O(\varepsilon)\,\|T\|_2$, and ε is the machine precision.

Note that if a 2 by 2 diagonal block is involved in the re-ordering, its off-diagonal elements are in general changed; the diagonal elements and the eigenvalues of the block are unchanged unless the block is sufficiently ill-conditioned, in which case they may be noticeably altered. It is possible for a 2 by 2 block to break into two 1 by 1 blocks, that is, for a pair of complex eigenvalues to become purely real.

The values of eigenvalues however are never changed by the re-ordering.

The approximate number of floating-point operations is

for real flavors:	$6n($`ifst`$-$`ilst`$)$	if *compq* = 'N';
	$12n($`ifst`$-$`ilst`$)$	if *compq* = 'V';
for complex flavors:	$20n($`ifst`$-$`ilst`$)$	if *compq* = 'N';
	$40n($`ifst`$-$`ilst`$)$	if *compq* = 'V'.

?trsen

Reorders the Schur factorization of a matrix and (optionally) computes the reciprocal condition numbers and invariant subspace for the selected cluster of eigenvalues.

Syntax

```
call strsen (job, compq, select, n, t, ldt, q, ldq, wr, wi, m, s,
   sep, work, lwork, iwork, liwork, info)
call dtrsen (job, compq, select, n, t, ldt, q, ldq, wr, wi, m, s,
   sep, work, lwork, iwork, liwork, info)
call ctrsen (job, compq, select, n, t, ldt, q, ldq, w, m, s,
   sep, work, lwork, info)
call ztrsen (job, compq, select, n, t, ldt, q, ldq, w, m, s,
   sep, work, lwork, info)
```

Description

This routine reorders the Schur factorization of a general matrix $A = QTQ^H$ so that a selected cluster of eigenvalues appears in the leading diagonal elements (or, for real flavors, diagonal blocks) of the Schur form.
The reordered Schur form R is computed by an unitary(orthogonal) similarity transformation: $R = Z^H TZ$. Optionally the updated matrix P of Schur vectors is computed as $P = QZ$, giving $A = PRP^H$.

Let

$$R = \begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{13} \end{bmatrix}$$

where the selected eigenvalues are precisely the eigenvalues of the leading m by m submatrix T_{11}. Let P be correspondingly partitioned as $(Q_1\ Q_2)$ where Q_1 consists of the first m columns of Q. Then $AQ_1 = Q_1 T_{11}$, and so the m columns of Q_1 form an orthonormal basis for the invariant subspace corresponding to the selected cluster of eigenvalues.

Optionally the routine also computes estimates of the reciprocal condition numbers of the average of the cluster of eigenvalues and of the invariant subspace.

Input Parameters

job — `CHARACTER*1`. Must be `'N'` or `'E'` or `'V'` or `'B'`.
If *job* = `'N'`, then no condition numbers are required.
If *job* = `'E'`, then only the condition number for the cluster of eigenvalues is computed.
If *job* = `'V'`, then only the condition number for the invariant subspace is computed.
If *job* = `'B'`, then condition numbers for both the cluster and the invariant subspace are computed.

compq — `CHARACTER*1`. Must be `'V'` or `'N'`.
If *compq* = `'V'`, then *Q* of the Schur vectors is updated.
If *compq* = `'N'`, then no Schur vectors are updated.

select — `LOGICAL`.
Array, `DIMENSION` at least max (1, *n*).
Specifies the eigenvalues in the selected cluster.
To select an eigenvalue λ_j, *select*(*j*) must be `.TRUE.` *For real flavors*: to select a complex conjugate pair of eigenvalues λ_j and λ_{j+1}(corresponding 2 by 2 diagonal block), *select*(*j*) and/or *select*(*j*+1) must be `.TRUE.`; the complex conjugate λ_j and λ_{j+1} must be either both included in the cluster or both excluded.

n — `INTEGER`. The order of the matrix *T* ($n \ge 0$).

t, q, work — `REAL` for `strsen`
`DOUBLE PRECISION` for `dtrsen`
`COMPLEX` for `ctrsen`
`DOUBLE COMPLEX` for `ztrsen`.
Arrays:
t (*ldt*,*) The *n* by *n* *T*.
The second dimension of *t* must be at least max(1, *n*).

q (*ldq*,*)
If *compq* = `'V'`, then *q* must contain *Q* of Schur vectors.
If *compq* = `'N'`, then *q* is not referenced.
The second dimension of *q* must be at least max(1, *n*) if *compq* = `'V'` and at least 1 if *compq* = `'N'`.

`work` (`lwork`) is a workspace array.
For complex flavors: the array `work` is not referenced if `job` = 'N'.
The actual amount of workspace required cannot exceed $n^2/4$ if `job` = 'E' or $n^2/2$ if `job` = 'V' or 'B'.

ldt — INTEGER. The first dimension of *t*; at least max(1, *n*).

ldq — INTEGER. The first dimension of *q*;
If *compq* = 'N', then *ldq* ≥ 1.
If *compq* = 'V', then *ldq* ≥ max(1,*n*).

lwork — INTEGER. The dimension of the array *work*.
If *job* = 'V' or 'B', *lwork* ≥ max(1,2*m*(*n*−*m*)).
If *job* = 'E', then *lwork* ≥ max(1,*m*(*n*−*m*))
If *job* = 'N', then *lwork* ≥ 1 for complex flavors and *lwork* ≥ max(1,*n*) for real flavors.

iwork — INTEGER.
iwork(*liwork*) is a workspace array.
The array *iwork* is not referenced if *job* = 'N' or 'E'.
The actual amount of workspace required cannot exceed $n^2/2$ if *job* = 'V' or 'B'.

liwork — INTEGER.
The dimension of the array *iwork*.
If *job* = 'V' or 'B', *liwork* ≥ max(1,2*m*(*n*−*m*)).
If *job* = 'E' or 'E', *liwork* ≥ 1.

Output Parameters

t — Overwritten by the updated matrix *R*.

q — If *compq* = 'V', *q* contains the updated matrix of Schur vectors; the first m columns of the *Q* form an orthogonal basis for the specified invariant subspace.

w — COMPLEX for ctrsen
DOUBLE COMPLEX for ztrsen.
Array, DIMENSION at least max(1,*n*).
The recorded eigenvalues of *R*. The eigenvalues are stored in the same order as on the diagonal of *R*.

wr, wi — REAL for strsen
DOUBLE PRECISION for dtrsen
Arrays, DIMENSION at least max(1,*n*).
Contain the real and imaginary parts, respectively, of the reordered eigenvalues

of R. The eigenvalues are stored in the same order as on the diagonal of R. Note that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.

m — `INTEGER`.
For complex flavors: the number of the specified invariant subspaces, which is the same as the number of selected eigenvalues (see *select*).
For real flavors: the dimension of the specified invariant subspace. The value of *m* is obtained by counting 1 for each selected real eigenvalue and 2 for each selected complex conjugate pair of eigenvalues (see *select*).

Constraint: $0 \le m \le n$.

s — `REAL` for single-precision flavors
`DOUBLE PRECISION` for double-precision flavors.
If *job* = `'E'` or `'B'`, *s* is a lower bound on the reciprocal condition number of the average of the selected cluster of eigenvalues. If $m = 0$ or n, then $s = 1$.
For real flavors: if *info* = 1, then *s* is set to zero.
s is not referenced if *job* = `'N'` or `'V'`.

sep — `REAL` for single-precision flavors
`DOUBLE PRECISION` for double-precision flavors.
If *job* = `'V'` or `'B'`, *sep* is the estimated reciprocal condition number of the specified invariant subspace.
If $m = 0$ or n, then $sep = \|T\|$.
For real flavors: if *info* = 1, then *sep* is set to zero.
sep is not referenced if *job* = `'N'` or `'E'`.

work(1) — On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

iwork(1) — On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

The computed matrix R is exactly similar to a matrix $T + E$, where
$\|E\|_2 = O(\varepsilon)\|T\|_2$, and ε is the machine precision.
The computed *s* cannot underestimate the true reciprocal condition number by more than a factor of $(\min(m, n-m))^{1/2}$; *sep* may differ from the true value by $(m*n-m^2)^{1/2}$. The angle between the

computed invariant subspace and the true subspace is $O(\varepsilon)\ \|A\|_2/sep$.
Note that if a 2 by 2 diagonal block is involved in the re-ordering, its off-diagonal elements are in general changed; the diagonal elements and the eigenvalues of the block are unchanged unless the block is sufficiently ill-conditioned, in which case they may be noticeably altered. It is possible for a 2 by 2 block to break into two 1 by 1 blocks, that is, for a pair of complex eigenvalues to become purely real. The values of eigenvalues however are never changed by the re-ordering.

?trsyl

Solves Sylvester's equation for real quasi-triangular or complex triangular matrices.

Syntax

```
call strsyl ( trana,tranb,isgn,m,n,a,lda,b,ldb,c,ldc,scale,info )
call dtrsyl ( trana,tranb,isgn,m,n,a,lda,b,ldb,c,ldc,scale,info )
call ctrsyl ( trana,tranb,isgn,m,n,a,lda,b,ldb,c,ldc,scale,info )
call ztrsyl ( trana,tranb,isgn,m,n,a,lda,b,ldb,c,ldc,scale,info )
```

Description

This routine solves the Sylvester matrix equation $\mathrm{op}(A)X \pm X\mathrm{op}(B) = \alpha C$, where $\mathrm{op}(A) = A$ or A^H, and the matrices A and B are upper triangular (or, for real flavors, upper quasi-triangular in canonical Schur form); $\alpha \le 1$ is a scale factor determined by the routine to avoid overflow in X; A is *m* by *m*, B is *n* by *n*, and C and X are both *m* by *n*. The matrix X is obtained by a straightforward process of back substitution.

The equation has a unique solution if and only if $\alpha_i \pm \beta_i \ne 0$, where $\{\alpha_i\}$ and $\{\beta_i\}$ are the eigenvalues of A and B, respectively, and the sign (+ or –) is the same as that used in the equation to be solved.

Input Parameters

trana — `CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`.
If *trana* = `'N'`, then $\mathrm{op}(A) = A$.
If *trana* = `'T'`, then $\mathrm{op}(A) = A^T$ (real flavors only).
If *trana* = `'C'` then $\mathrm{op}(A) = A^H$.

tranb	`CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`. If *tranb* = `'N'`, then op(B) = B. If *tranb* = `'T'`, then op(B) = B^T (real flavors only). If *tranb* = `'C'`, then op(B) = B^H.
isgn	`INTEGER`. Indicates the form of the Sylvester equation. If *isgn* = +1, op(A)X + Xop(B) = αC. If *isgn* = –1, op(A)X – Xop(B) = αC.
m	`INTEGER`. The order of A, and the number of rows in X and C (*m* ≥ 0).
n	`INTEGER`. The order of B, and the number of columns in X and C (*n* ≥ 0).
a, b, c	`REAL` for `strsyl` `DOUBLE PRECISION` for `dtrsyl` `COMPLEX` for `ctrsyl` `DOUBLE COMPLEX` for `ztrsyl`. Arrays: *a*(*lda*,*) contains the matrix A. The second dimension of *a* must be at least max(1, *m*). *b*(*ldb*,*) contains the matrix B. The second dimension of *b* must be at least max(1, *n*). *c*(*ldc*,*) contains the matrix C. The second dimension of *c* must be at least max(1, *n*).
lda	`INTEGER`. The first dimension of *a*; at least max(1, *m*).
ldb	`INTEGER`. The first dimension of *b*; at least max(1, *n*).
ldc	`INTEGER`. The first dimension of *c*; at least max(1, *n*).

Output Parameters

c	Overwritten by the solution matrix X.
scale	`REAL` for single-precision flavors `DOUBLE PRECISION` for double-precision flavors. The value of the scale factor α.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = 1, A and B have common or close eigenvalues perturbed values were used to solve the equation.

Application Notes

Let X be the exact, Y the corresponding computed solution, and R the residual matrix: $R = C - (AY \pm YB)$. Then the residual is always small:

$$||R||_F = O(\varepsilon)\ (||A||_F + ||B||_F)\ ||Y||_F .$$

However, Y is not necessarily the exact solution of a slightly perturbed equation; in other words, the solution is not backwards stable.

For the forward error, the following bound holds:

$$||Y - X||_F \le ||R||_F / \mathrm{sep}(A, B)$$

but this may be a considerable overestimate. See [Golub96] for a definition of sep(A, B).

The approximate number of floating-point operations for real flavors is $m*n*(m + n)$. For complex flavors it is 4 times greater.

Generalized Nonsymmetric Eigenvalue Problems

This section describes LAPACK routines for solving generalized nonsymmetric eigenvalue problems, reordering the generalized Schur factorization of a pair of matrices, as well as performing a number of related computational tasks.

A *generalized nonsymmetric eigenvalue problem* is as follows: given a pair of nonsymmetric (or non-Hermitian) n-by-n matrices A and B, find the *generalized eigenvalues* λ and the corresponding *generalized eigenvectors* x and y that satisfy the equations

$$Ax = \lambda Bx \quad \text{(right generalized eigenvectors } x\text{)}$$

and

$$y^H A = \lambda y^H B \quad \text{(left generalized eigenvectors } y\text{).}$$

Table 4-6 lists LAPACK routines used to solve the generalized nonsymmetric eigenvalue problems and the generalized Sylvester
equation.

Table 4-6 Computational Routines for Solving Generalized Nonsymmetric Eigenvalue Problems

Routine name	Operation performed
?gghrd	Reduces a pair of matrices to generalized upper Hessenberg form using orthogonal/unitary transformations.
?ggbal	Balances a pair of general real or complex matrices.
?ggbak	Forms the right or left eigenvectors of a generalized eigenvalue problem.
?hgeqz	Implements the QZ method for finding the generalized eigenvalues of the matrix pair (H,T).
?tgevc	Computes some or all of the right and/or left generalized eigenvectors of a pair of upper triangular matrices
?tgexc	Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that one diagonal block of (A,B) moves to another row index.
?tgsen	Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that a selected cluster of eigenvalues appears in the leading diagonal blocks of (A,B).
?tgsyl	Solves the generalized Sylvester equation.
?tgsna	Estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a pair of matrices in generalized real Schur canonical form.

?gghrd

Reduces a pair of matrices to generalized upper Hessenberg form using orthogonal/unitary transformations.

Syntax

```
call sgghrd ( compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq,
              z, ldz, info )
call dgghrd ( compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq,
              z, ldz, info )
call cgghrd ( compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq,
              z, ldz, info )
call zgghrd ( compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq,
              z, ldz, info )
```

Description

This routine reduces a pair of real/complex matrices (A,B) to generalized upper Hessenberg form using orthogonal/unitary transformations, where A is a general matrix and B is upper triangular. The form of the generalized eigenvalue problem is $Ax = \lambda Bx$, and B is typically made upper triangular by computing its QR factorization and moving the orthogonal matrix Q to the left side of the equation.

This routine simultaneously reduces A to a Hessenberg matrix H:

$$Q^H A \; Z = H$$

and transforms B to another upper triangular matrix T:

$$Q^H B \; Z = T$$

in order to reduce the problem to its standard form $Hy = \lambda Ty$ where $y = Z^H x$.

The orthogonal/unitary matrices Q and Z are determined as products of Givens rotations. They may either be formed explicitly, or they may be postmultiplied into input matrices Q_1 and Z_1, so that

$$Q_1 \; A \; Z_1^H = (Q_1 Q) \; H \; (Z_1 Z)^H$$

$$Q_1 \, B \; Z_1^H = (Q_1 Q) \; T \; (Z_1 Z)^H$$

If Q_1 is the orthogonal matrix from the QR factorization of B in the original equation $Ax = \lambda Bx$, then `?gghrd` reduces the original problem to generalized Hessenberg form.

Input Parameters

compq — CHARACTER*1. Must be 'N', 'I', or 'V'.
If *compq* = 'N', matrix Q is not computed.
If *compq* = 'I', Q is initialized to the unit matrix, and the orthogonal/unitary matrix Q is returned;
If *compq* = 'V', Q must contain an orthogonal/unitary matrix Q_1 on entry, and the product Q_1Q is returned.

compz — CHARACTER*1. Must be 'N', 'I', or 'V'.
If *compz* = 'N', matrix Z is not computed.
If *compz* = 'I', Z is initialized to the unit matrix, and the orthogonal/unitary matrix Z is returned;
If *compz* = 'V', Z must contain an orthogonal/unitary matrix Z_1 on entry, and the product Z_1Z is returned.

n — INTEGER. The order of the matrices A and B ($n \ge 0$).

ilo, ihi — INTEGER. *ilo* and *ihi* mark the rows and columns of A which are to be reduced. It is assumed that A is already upper triangular in rows and columns 1:*ilo*-1 and *ihi*+1:*n*. Values of *ilo* and *ihi* are normally set by a previous call to ?ggbal; otherwise they should be set to 1 and *n* respectively.
Constraint:
If $n > 0$, then $1 \le ilo \le ihi \le n$;
if $n = 0$, then $ilo = 1$ and $ihi = 0$.

a, b, q, z — REAL for sgghrd
DOUBLE PRECISION for dgghrd
COMPLEX for cgghrd
DOUBLE COMPLEX for zgghrd.
Arrays:
a(*lda*,*) contains the *n*-by-*n* general matrix A.
The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the *n*-by-*n* upper triangular matrix B.
The second dimension of *b* must be at least max(1, *n*).

q (*ldq*,*)
If *compq* ='N', then *q* is not referenced.
If *compq* ='I', then, on entry, *q* need not be set.
If *compq* ='V', then *q* must contain the orthogonal/unitary matrix Q_1, typically from the QR factorization of B.
The second dimension of *q* must be at least max(1, *n*).

z (*ldz*,*)
If *compq* ='N', then *z* is not referenced.
If *compq* ='I', then, on entry, *z* need not be set.
If *compq* ='V', then *z* must contain the orthogonal/unitary matrix Z_1.
The second dimension of *z* must be at least max(1, *n*).

lda INTEGER. The first dimension of *a*; at least max(1, *n*).

ldb INTEGER. The first dimension of *b*; at least max(1, *n*).

ldq INTEGER. The first dimension of *q*;
If *compq* ='N', then *ldq* ≥ 1.
If *compq* ='I'or 'V', then *ldq* ≥ max(1,*n*).

ldz INTEGER. The first dimension of *z*;
If *compq* ='N', then *ldz* ≥ 1.
If *compq* ='I'or 'V', then *ldz* ≥ max(1,*n*).

Output Parameters

a On exit, the upper triangle and the first subdiagonal of *A* are overwritten with the upper Hessenberg matrix *H*, and the rest is set to zero.

b On exit, overwritten by the upper triangular matrix
$T = Q^H B\ Z$. The elements below the diagonal are set to zero.

q If *compq* ='I', then *q* contains the orthogonal/unitary matrix *Q*, where Q^H is the product of the Givens transformations which are applied to *A* and *B* on the left;
If *compq* ='V', then *q* is overwritten by the
product Q_1Q.

z If *compq* ='I', then *z* contains the orthogonal/unitary matrix *Z*, which is the product of the Givens transformations which are applied to *A* and *B* on the right;
If *compq* ='V', then *z* is overwritten by the
product Z_1Z.

info INTEGER.
If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

?ggbal

Balances a pair of general real or complex matrices.

Syntax

```
call sggbal ( job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale,
              work, info )
call dggbal ( job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale,
              work, info )
call cggbal ( job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale,
              work, info )
call zggbal ( job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale,
              work, info )
```

Description

This routine balances a pair of general real/complex matrices (A,B). This involves, first, permuting A and B by similarity transformations to isolate eigenvalues in the first 1 to `ilo`-1 and last `ihi`+1 to `n` elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns `ilo` to `ihi` to make the rows and columns as close in norm as possible. Both steps are optional.
Balancing may reduce the 1-norm of the matrices, and improve the accuracy of the computed eigenvalues and/or eigenvectors in the generalized eigenvalue problem $Ax = \lambda Bx$.

Input Parameters

job — `CHARACTER*1`. Specifies the operations to be performed on A and B. Must be `'N'` or `'P'` or `'S'` or `'B'`.
If *job* = `'N'`, then no operations are done; simply set *ilo*=1, *ihi*=*n*, *lscale*(i) =1.0 and *rscale*(i)=1.0 for
i = 1,...,*n*.
If *job* = `'P'`, then permute only.
If *job* = `'S'`, then scale only.
If *job* = `'B'`, then both permute and scale.

n — `INTEGER`. The order of the matrices A and B ($n \ge 0$).

a, b — `REAL` for `sggbal`
`DOUBLE PRECISION` for `dggbal`
`COMPLEX` for `cggbal`

	`DOUBLE COMPLEX` for `zggbal`. Arrays: *a*(*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the matrix *B*. The second dimension of *b* must be at least max(1, *n*).
lda	`INTEGER`. The first dimension of *a*; at least max(1, *n*).
ldb	`INTEGER`. The first dimension of *b*; at least max(1, *n*).
work	`REAL` for single precision flavors `DOUBLE PRECISION` for double precision flavors. Workspace array, `DIMENSION` at least max(1, 6*n*).

Output Parameters

a, *b*	Overwritten by the balanced matrices *A* and *B*, respectively. If *job* = 'N', *a* and *b* are not referenced.
ilo, *ihi*	`INTEGER`. *ilo* and *ihi* are set to integers such that on exit *a*(*i*,*j*)=0 and *b*(*i*,*j*)=0 if *i*>*j* and *j*=1,...,*ilo*-1 or *i*=*ihi*+1,...,*n*. If *job* = 'N' or 'S', then *ilo* = 1 and *ihi* = *n*.
lscale, *rscale*	`REAL` for single precision flavors `DOUBLE PRECISION` for double precision flavors. Arrays, `DIMENSION` at least max(1, *n*). *lscale* contains details of the permutations and scaling factors applied to the left side of *A* and *B*. If P_j is the index of the row interchanged with row *j*, and D_j is the scaling factor applied to row *j*, then *lscale*(*j*) = P_j , for *j* = 1,..., *ilo*-1 = D_j , for *j* = *ilo*,...,*ihi* = P_j , for *j* = *ihi*+1,..., *n*. *rscale* contains details of the permutations and scaling factors applied to the right side of *A* and *B*. If P_j is the index of the column interchanged with column *j*, and D_j is the scaling factor applied to column *j*, then

rscale(*j*) = P_j , for *j* = 1,..., *ilo*-1
= D_j , for *j* = *ilo*,...,*ihi*
= P_j , for *j* = *ihi*+1,..., *n*

The order in which the interchanges are made is *n* to *ihi*+1, then 1 to *ilo*-1.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

?ggbak

Forms the right or left eigenvectors of a generalized eigenvalue problem.

Syntax

```
call sggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call dggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call cggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call zggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
```

Description

This routine forms the right or left eigenvectors of a real/complex generalized eigenvalue problem

$$Ax = \lambda Bx$$

by backward transformation on the computed eigenvectors of the balanced pair of matrices output by `?ggbal`.

Input Parameters

job — `CHARACTER*1`. Specifies the type of backward transformation required. Must be `'N'`, `'P'`, `'S'`, or `'B'`.
If *job* = `'N'`, then no operations are done; return.
If *job* = `'P'`, then do backward transformation for permutation only.
If *job* = `'S'`, then do backward transformation for scaling only.
If *job* = `'B'`, then do backward transformation for both permutation and scaling.
This argument must be the same as the argument *job* supplied to `?ggbal`.

side — `CHARACTER*1`. Must be `'L'` or `'R'`.
If *side* = `'L'`, then *v* contains left eigenvectors .
If *side* = `'R'`, then *v* contains right eigenvectors .

n — `INTEGER`. The number of rows of the matrix V ($n \geq 0$).

ilo, ihi — `INTEGER`. The integers *ilo* and *ihi* determined by `?gebal`. Constraint:
If $n > 0$, then $1 \leq ilo \leq ihi \leq n$;
if $n = 0$, then $ilo = 1$ and $ihi = 0$.

lscale, rscale REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(1, *n*).

The array *lscale* contains details of the permutations and/or scaling factors applied to the left side of *A* and *B*, as returned by ?ggbal.

The array *rscale* contains details of the permutations and/or scaling factors applied to the right side of *A* and *B*, as returned by ?ggbal.

m INTEGER. The number of columns of the matrix *V* ($m \ge 0$).

v REAL for sggbak
DOUBLE PRECISION for dggbak
COMPLEX for cggbak
DOUBLE COMPLEX for zggbak.
Array *v*(*ldv*,*). Contains the matrix of right or left eigenvectors to be transformed, as returned by ?tgevc.
The second dimension of *v* must be at least max(1, *m*).

ldv INTEGER. The first dimension of *v*; at least max(1, *n*).

Output Parameters

v Overwritten by the transformed eigenvectors

info INTEGER.
If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

?hgeqz

Implements the QZ method for finding the generalized eigenvalues of the matrix pair (H,T).

Syntax

```
call shgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alphar,
            alphai, beta, q, ldq, z, ldz, work, lwork, info )
call dhgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alphar,
            alphai, beta, q, ldq, z, ldz, work, lwork, info )
call chgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alpha,
            beta, q, ldq, z, ldz, work, lwork, rwork, info )
call zhgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alpha,
            beta, q, ldq, z, ldz, work, lwork, rwork, info )
```

Description

This routine computes the eigenvalues of a real/complex matrix pair (H,T), where H is an upper Hessenberg matrix and T is upper triangular, using the double-shift version (for real flavors) or single-shift version (for complex flavors) of the QZ method.
Matrix pairs of this type are produced by the reduction to generalized upper Hessenberg form of a real/complex matrix pair (A,B):

$$A = Q_1 \; H \; Z_1^H , \qquad B = Q_1 \; T \; Z_1^H ,$$

as computed by `?gghrd`.

For real flavors:
If `job`=`'S'`, then the Hessenberg-triangular pair (H,T) is also reduced to generalized Schur form,

$$H = Q \; S \; Z^T , \qquad T = Q \; P \; Z^T ,$$

where Q and Z are orthogonal matrices, P is an upper triangular matrix, and S is a quasi-triangular matrix with 1-by-1 and 2-by-2 diagonal blocks.
The 1-by-1 blocks correspond to real eigenvalues of the matrix pair (H,T) and the 2-by-2 blocks correspond to complex conjugate pairs of eigenvalues.
Additionally, the 2-by-2 upper triangular diagonal blocks of P corresponding to 2-by-2 blocks of S are reduced to positive diagonal form, that is, if S(j+1,j) is non-zero, then P(j+1,j) = P(j,j+1) = 0, P(j,j) > 0, and P(j+1,j+1) > 0.

For complex flavors:
If `job` = `'S'`, then the Hessenberg-triangular pair (H,T) is also reduced to generalized Schur form,

$$H = Q\ S\ Z^H, \qquad T = Q\ P\ Z^H,$$

where Q and Z are unitary matrices, and S and P are upper triangular.

For all function flavors:
Optionally, the orthogonal/unitary matrix Q from the generalized Schur factorization may be postmultiplied into an input matrix Q_1, and the orthogonal/unitary matrix Z may be postmultiplied into an input matrix Z_1. If Q_1 and Z_1 are the orthogonal/unitary matrices from `?gghrd` that reduced the matrix pair (A,B) to generalized upper Hessenberg form, then the output matrices Q_1Q and Z_1Z are the orthogonal/unitary factors from the generalized Schur factorization of (A,B):

$$A = (Q_1Q)\ S\ (Z_1Z)^H, \qquad B = (Q_1Q)\ P\ (Z_1Z)^H.$$

To avoid overflow, eigenvalues of the matrix pair (H,T) (equivalently, of (A,B)) are computed as a pair of values (*alpha,beta*). For `chgeqz/zhgeqz`, *alpha* and *beta* are complex, and for `shgeqz/dhgeqz`, *alpha* is complex and *beta* real. If *beta* is nonzero, λ = *alpha* / *beta* is an eigenvalue of the generalized nonsymmetric eigenvalue problem (GNEP)

$$Ax = \lambda Bx$$

and if *alpha* is nonzero, μ = *beta* / *alpha* is an eigenvalue of the alternate form of the GNEP

$$\mu Ay = By .$$

Real eigenvalues (for real flavors) or the values of *alpha* and *beta* for the i-th eigenvalue (for complex flavors) can be read directly from the generalized Schur form:

alpha = S(i,i), *beta* = P(i,i).

Input Parameters

`job` — `CHARACTER*1`. Specifies the operations to be performed. Must be `'E'` or `'S'`.
If *`job`* = `'E'`, then compute eigenvalues only;
If *`job`* = `'S'`, then compute eigenvalues and the Schur form.

`compq` — `CHARACTER*1`. Must be `'N'`, `'I'`, or `'V'`.
If *`compq`* = `'N'`, left Schur vectors (*q*) are not computed;
If *`compq`* = `'I'`, *`q`* is initialized to the unit matrix and the matrix of left Schur vectors of (H,T) is returned;
If *`compq`* = `'V'`, *`q`* must contain an orthogonal/unitary matrix Q_1 on entry and the product Q_1Q is returned.

`compz` — `CHARACTER*1`. Must be `'N'`, `'I'`, or `'V'`.
If *`compz`* = `'N'`, left Schur vectors (*q*) are not computed;
If *`compz`* = `'I'`, *`z`* is initialized to the unit matrix and the matrix of right Schur vectors of (H,T) is returned;

If *compz* = 'V', *z* must contain an orthogonal/unitary matrix Z_1 on entry and the product Z_1Z is returned.

n — INTEGER. The order of the matrices *H, T, Q,* and *Z* ($n \geq 0$).

ilo, ihi — INTEGER. *ilo* and *ihi* mark the rows and columns of *H* which are in Hessenberg form. It is assumed that *H* is already upper triangular in rows and columns 1:*ilo*-1 and *ihi*+1:*n*. Constraint:
If $n > 0$, then $1 \leq ilo \leq ihi \leq n$;
if $n = 0$, then $ilo = 1$ and $ihi = 0$.

h,t,q,z,work — REAL for shgeqz
DOUBLE PRECISION for dhgeqz
COMPLEX for chgeqz
DOUBLE COMPLEX for zhgeqz.
Arrays:
On entry, *h*(*ldh*,*) contains the *n*-by-*n* upper Hessenberg matrix *H*.
The second dimension of *h* must be at least max(1, *n*).

On entry, *t*(*ldt*,*) contains the *n*-by-*n* upper triangular matrix *T*.
The second dimension of *t* must be at least max(1, *n*).

q (*ldq*,*):
On entry, if *compq* ='V', this array contains the orthogonal/unitary matrix Q_1 used in the reduction of (*A,B*) to generalized Hessenberg form.
If *compq* ='N', then *q* is not referenced.
The second dimension of *q* must be at least max(1, *n*).

z (*ldz*,*):
On entry, if *compz* ='V', this array contains the orthogonal/unitary matrix Z_1 used in the reduction of (*A,B*) to generalized Hessenberg form.
If *compz* ='N', then *z* is not referenced.
The second dimension of *z* must be at least max(1, *n*).

work(*lwork*) is a workspace array.

ldh — INTEGER. The first dimension of *h*; at least max(1, *n*).

ldt — INTEGER. The first dimension of *t*; at least max(1, *n*).

ldq — INTEGER. The first dimension of *q*;
If *compq* ='N', then $ldq \geq 1$.
If *compq* ='I' or 'V', then $ldq \geq \max(1,n)$.

ldz — INTEGER. The first dimension of *z*;
If *compq* = 'N', then *ldz* ≥ 1.
If *compq* = 'I' or 'V', then *ldz* ≥ max(1,*n*).

lwork — INTEGER. The dimension of the array *work*.
lwork ≥ max(1,*n*).

rwork — REAL for chgeqz
DOUBLE PRECISION for zhgeqz.
Workspace array, DIMENSION at least max(1, *n*). Used in complex flavors only.

Output Parameters

h — *For real flavors*:
If *job* = 'S', then, on exit, *h* contains the upper quasi-triangular matrix *S* from the generalized Schur factorization; 2-by-2 diagonal blocks (corresponding to complex conjugate pairs of eigenvalues) are returned in standard form, with
h(i,i) = *h*(i+1, i+1) and
h(i+1, i) * *h*(i, i+1) < 0.
If *job* = 'E', then on exit the diagonal blocks of *h* match those of *S*, but the rest of *h* is unspecified.

For complex flavors:
If *job* = 'S', then, on exit, *h* contains the upper triangular matrix *S* from the generalized Schur factorization.
If *job* = 'E', then on exit the diagonal of *h* matches that of *S*, but the rest of *h* is unspecified.

t — If *job* = 'S', then, on exit, *t* contains the upper triangular matrix *P* from the generalized Schur factorization.
For real flavors:
2-by-2 diagonal blocks of *P* corresponding to 2-by-2 blocks of *S* are reduced to positive diagonal form, that is, if *h*(j+1,j) is non-zero, then
t(j+1,j)=*t*(j,j+1)=0 and *t*(j,j) and *t*(j+1,j+1) will be positive.

If *job* = 'E', then on exit the diagonal blocks of *t* match those of *P*, but the rest of *t* is unspecified.

For complex flavors:
If *job* = 'E', then on exit the diagonal of *t* matches that of *P*, but the rest of *t* is unspecified.

alphar,alphai REAL for shgeqz;
DOUBLE PRECISION for dhgeqz.
Arrays, DIMENSION at least max(1,*n*).
The real and imaginary parts, respectively, of each scalar *alpha* defining an eigenvalue of GNEP.

If *alphai*(j) is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-th eigenvalues are a complex conjugate pair, with *alphai*(j+1) = -*alphai*(j).

alpha COMPLEX for chgeqz;
DOUBLE COMPLEX for zhgeqz.
Array, DIMENSION at least max(1,*n*).
The complex scalars *alpha* that define the eigenvalues of GNEP. *alphai*(i) = *S*(i,i) in the generalized Schur factorization.

beta REAL for shgeqz
DOUBLE PRECISION for dhgeqz
COMPLEX for chgeqz
DOUBLE COMPLEX for zhgeqz.
Array, DIMENSION at least max(1,*n*).
For real flavors:
The scalars *beta* that define the eigenvalues of GNEP.
Together, the quantities *alpha* = (*alphar*(j), *alphai*(j)) and *beta* = *beta*(j) represent the j-th eigenvalue of the matrix pair (*A,B*), in one of the forms λ = *alpha/beta* or μ = *beta/alpha* . Since either λ or μ may overflow, they should not, in general, be computed.

For complex flavors:
The real non-negative scalars *beta* that define the eigenvalues of GNEP.
beta(i) = *P*(i,i) in the generalized Schur factorization.
Together, the quantities *alpha* = *alpha*(j) and *beta* = *beta*(j) represent the j-th eigenvalue of the matrix pair (*A,B*), in one of the forms λ = *alpha/beta* or
μ = *beta/alpha* . Since either λ or μ may overflow, they should not, in general, be computed.

q On exit, if *compq* = 'I', *q* is overwritten by the orthogonal/unitary matrix of left Schur vectors of the pair (*H,T*), and if *compq* = 'V', *q* is overwritten by the orthogonal/unitary matrix of left Schur vectors of (*A,B*).

z On exit, if *compz* = 'I', *z* is overwritten by the orthogonal/unitary matrix of right Schur vectors of the pair (*H,T*), and if *compz* = 'V', *z* is overwritten by the orthogonal/unitary matrix of right Schur vectors of (*A,B*).

work(1)	If *info* ≥ 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = 1,...,*n*, the *QZ* iteration did not converge. (*H,T*) is not in Schur form, but *alphar*(i), *alphai*(i) (for real flavors), *alpha*(i) (for complex flavors), and *beta*(i), i=*info*+1,...,*n* should be correct. If *info* = *n*+1,...,2*n*, the shift calculation failed. (*H,T*) is not in Schur form, but *alphar*(i), *alphai*(i) (for real flavors), *alpha*(i) (for complex flavors), and *beta*(i), i =*info*-*n*+1,...,*n* should be correct.

?tgevc

Computes some or all of the right and/or left generalized eigenvectors of a pair of upper triangular matrices.

Syntax

```
call stgevc ( side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr,
              ldvr, mm, m, work, info )
call dtgevc ( side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr,
              ldvr, mm, m, work, info )
call ctgevc ( side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr,
              ldvr, mm, m, work, rwork, info )
call ztgevc ( side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr,
              ldvr, mm, m, work, rwork, info )
```

Description

This routine computes some or all of the right and/or left eigenvectors of a pair of real/complex matrices (*S,P*), where *S* is quasi-triangular (for real flavors) or upper triangular (for complex flavors) and *P* is upper triangular.

Matrix pairs of this type are produced by the generalized Schur factorization of a real/complex matrix pair (*A,B*):

$$A = Q\,S\,Z^H \quad , \qquad B = Q\,P\,Z^H$$

as computed by `?gghrd` plus `?hgeqz`.

The right eigenvector *x* and the left eigenvector *y* of (*S,P*) corresponding to an eigenvalue *w* are defined by:

$$S\,x = w\,P\,x \ , \qquad y^H S = w\,y^H P$$

The eigenvalues are not input to this routine, but are computed directly from the diagonal blocks or diagonal elements of *S* and *P*.

This routine returns the matrices *X* and/or *Y* of right and left eigenvectors of (*S,P*), or the products *Z X* and/or *Q Y*, where *Z* and *Q* are input matrices.
If *Q* and *Z* are the orthogonal/unitary factors from the generalized Schur factorization of a matrix pair (*A,B*), then *Z X* and *Q Y* are the matrices of right and left eigenvectors of (*A,B*).

Input Parameters

side — `CHARACTER*1`. Must be `'R'`, `'L'`, or `'B'`.
If *side* = `'R'`, compute right eigenvectors only.
If *side* = `'L'`, compute left eigenvectors only.
If *side* = `'B'`, compute both right and left eigenvectors.

howmny — `CHARACTER*1`. Must be `'A'` , `'B'`, or `'S'`.
If *howmny* =`'A'`, compute all right and/or left eigenvectors.
If *howmny* =`'B'`, compute all right and/or left eigenvectors, backtransformed by the matrices in *vr* and/or *vl*.
If *howmny* =`'S'`, compute selected right and/or left eigenvectors, specified by the logical array *select*.

select — `LOGICAL`.
Array, `DIMENSION` at least max (1, *n*).
If *howmny* =`'S'` , *select* specifies the eigenvectors to be computed.
If *howmny*= `'A'`or `'B'`, *select* is not referenced.
For real flavors:
If ω_j is a real eigenvalue, the corresponding real eigenvector is computed if *select*(j) is `.TRUE.`.
If ω_j and ω_{j+1} are the real and imaginary parts of a complex eigenvalue, the corresponding complex eigenvector is computed if either *select*(j) or *select*(j+1) is `.TRUE.` , and on exit *select*(j) is set to `.TRUE.`and *select*(j+1) is set to `.FALSE.`.
For complex flavors:
The eigenvector corresponding to the j-th eigenvalue is computed if *select*(j) is `.TRUE.`.

n — `INTEGER`. The order of the matrices A and B ($n \geq 0$).

s, *p*, *vl*, *vr*, *work* — `REAL` for `stgevc`
`DOUBLE PRECISION` for `dtgevc`
`COMPLEX` for `ctgevc`
`DOUBLE COMPLEX` for `ztgevc`.
Arrays:

s(*lds*,*) contains the matrix S from a generalized Schur factorization as computed by `?hgeqz`. This matrix is upper quasi-triangular for real flavors, and upper triangular for complex flavors.
The second dimension of *s* must be at least max(1, *n*).

p(*ldp*,*) contains the upper triangular matrix *P* from a generalized Schur factorization as computed by ?hgeqz.
For real flavors, 2-by-2 diagonal blocks of *P* corresponding to 2-by-2 blocks of *S* must be in positive diagonal form.
For complex flavors, *P* must have real diagonal elements.
The second dimension of *p* must be at least max(1, *n*).

If *side* ='L' or 'B' and *howmny* ='B',
vl(*ldvl*,*) must contain an *n*-by-*n* matrix *Q* (usually the orthogonal/unitary matrix *Q* of left Schur vectors returned by ?hgeqz). The second dimension of *vl* must be at least max(1, *mm*). If *side* ='R' , *vl* is not referenced.

If *side* ='R' or 'B' and *howmny* ='B',
vr(*ldvr*,*) must contain an *n*-by-*n* matrix *Z* (usually the orthogonal/unitary matrix *Z* of right Schur vectors returned by ?hgeqz). The second dimension of *vr* must be at least max(1, *mm*). If *side* ='L' , *vr* is not referenced.

work(*) is a workspace array.
DIMENSION at least max (1, 6*n) for real flavors and at least max (1, 2*n) for complex flavors.

lda INTEGER. The first dimension of *a*; at least max(1, *n*).

ldb INTEGER. The first dimension of *b*; at least max(1, *n*).

ldvl INTEGER. The first dimension of *vl*;
If *side* ='L'or 'B', then *ldvl* ≥ max(1,*n*).
If *side* ='R', then *ldvl* ≥ 1.

ldvr INTEGER. The first dimension of *vr*;
If *side* ='R'or 'B', then *ldvr* ≥ max(1,*n*).
If *side* ='L', then *ldvr* ≥ 1.

mm INTEGER. The number of columns in the arrays *vl* and/or *vr* (*mm* ≥ *m*).

rwork REAL for ctgevc
DOUBLE PRECISION for ztgevc.
Workspace array, DIMENSION at least max (1, 2*n). Used in complex flavors only.

Output Parameters

vl On exit, if *side* ='L'or 'B', *vl* contains:
if *howmny* ='A', the matrix *Y* of left eigenvectors of (*S,P*);
if *howmny* ='B', the matrix *QY*;
if *howmny* ='S', the left eigenvectors of (*S,P*) specified by *select*, stored

consecutively in the columns of `vl`, in the same order as their eigenvalues.
For real flavors:
A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.

vr — On exit, if *side* ='R'or 'B', *vr* contains:
if *howmny* ='A', the matrix X of right eigenvectors of (S,P);
if *howmny* ='B', the matrix ZX;
if *howmny* ='S', the right eigenvectors of (S,P) specified by *select*, stored consecutively in the columns of *vr*, in the same order as their eigenvalues.
For real flavors:
A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.

m — INTEGER. The number of columns in the arrays *vl* and/or *vr* actually used to store the eigenvectors.
If *howmny* ='A' or 'B', *m* is set to *n*.
For real flavors:
Each selected real eigenvector occupies one column and each selected complex eigenvector occupies two columns.
For complex flavors:
Each selected eigenvector occupies one column.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
For real flavors:
If *info* = *i*>0, the 2-by-2 block (*i*:*i*+1) does not have a complex eigenvalue.

?tgexc

Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that one diagonal block of (A,B) moves to another row index.

Syntax

```
call stgexc ( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz,
              ifst, ilst, work, lwork, info )
call dtgexc ( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz,
              ifst, ilst, work, lwork, info )
call ctgexc ( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz,
              ifst, ilst, info )
call ztgexc ( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz,
              ifst, ilst, info )
```

Description

This routine reorders the generalized real-Schur/Schur decomposition of a real/complex matrix pair (A,B) using an orthogonal/unitary equivalence transformation

$$(A, B) = Q \ (A, B) \ Z^H,$$

so that the diagonal block of (A, B) with row index `ifst` is moved to row `ilst`.
Matrix pair (A, B) must be in generalized real-Schur/Schur canonical form (as returned by `?gges`), i.e. A is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks and B is upper triangular.
Optionally, the matrices Q and Z of generalized Schur vectors are updated.

$$Q(\text{in}) * A(\text{in}) * Z(\text{in})' = Q(\text{out}) * A(\text{out}) * Z(\text{out})'$$

$$Q(\text{in}) * B(\text{in}) * Z(\text{in})' = Q(\text{out}) * B(\text{out}) * Z(\text{out})'.$$

Input Parameters

wantq, wantz `LOGICAL`.
If *wantq* = `.TRUE.`, update the left transformation matrix Q;
If *wantq* = `.FALSE.`, do not update Q;

	If *wantz* =.TRUE., update the right transformation matrix *Z*; If *wantz* =.FALSE., do not update *Z*.
n	INTEGER. The order of the matrices *A* and *B* ($n \ge 0$).
a, b, q, z	REAL for stgexc DOUBLE PRECISION for dtgexc COMPLEX for ctgexc DOUBLE COMPLEX for ztgexc. Arrays: *a*(*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the matrix *B*. The second dimension of *b* must be at least max(1, *n*). *q* (*ldq*,*) If *wantq* =.FALSE., then *q* is not referenced. If *wantq* =.TRUE., then *q* must contain the orthogonal/unitary matrix *Q*. The second dimension of *q* must be at least max(1, *n*). *z* (*ldz*,*) If *wantz* =.FALSE., then *z* is not referenced. If *wantz* =.TRUE., then *z* must contain the orthogonal/unitary matrix *Z*. The second dimension of *z* must be at least max(1, *n*).
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
ldb	INTEGER. The first dimension of *b*; at least max(1, *n*).
ldq	INTEGER. The first dimension of *q*; If *wantq* =.FALSE., then $ldq \ge 1$. If *wantq* =.TRUE., then $ldq \ge \max(1,n)$.
ldz	INTEGER. The first dimension of *z*; If *wantz* =.FALSE., then $ldz \ge 1$. If *wantz* =.TRUE., then $ldz \ge \max(1,n)$.
ifst, ilst	INTEGER. Specify the reordering of the diagonal blocks of (*A*, *B*). The block with row index *ifst* is moved to row *ilst*, by a sequence of swapping between adjacent blocks. Constraint: $1 \le ifst, ilst \le n$.
work	REAL for stgexc; DOUBLE PRECISION for dtgexc. Workspace array, DIMENSION (*lwork*). Used in real flavors only.

lwork INTEGER. The dimension of *work*; must be at least
$4n$ +16.

Output Parameters

a, b Overwritten by the updated matrices *A* and *B*.

ifst, ilst Overwritten for real flavors only.
If *ifst* pointed to the second row of a 2 by 2 block on entry, it is changed to point to the first row; *ilst* always points to the first row of the block in its final position (which may differ from its input value by ±1).

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.
If *info* = 1, the transformed matrix pair (*A*, *B*) would be too far from generalized Schur form; the problem is ill-conditioned. (*A*, *B*) may have been partially reordered, and *ilst* points to the first row of the current position of the block being moved.

?tgsen

Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that a selected cluster of eigenvalues appears in the leading diagonal blocks of (A,B).

Syntax

```
call stgsen ( ijob, wantq, wantz, select, n, a, lda, b, ldb, alphar,
              alphai, beta, q, ldq, z, ldz, m, pl, pr, dif, work,
               lwork, iwork, liwork, info )
call dtgsen ( ijob, wantq, wantz, select, n, a, lda, b, ldb, alphar,
              alphai, beta, q, ldq, z, ldz, m, pl, pr, dif, work,
               lwork, iwork, liwork, info )
call ctgsen ( ijob, wantq, wantz, select, n, a, lda, b, ldb, alpha,
              beta, q, ldq, z, ldz, m, pl, pr, dif, work,
               lwork, iwork, liwork, info )
call ztgsen ( ijob, wantq, wantz, select, n, a, lda, b, ldb, alpha,
              beta, q, ldq, z, ldz, m, pl, pr, dif, work,
               lwork, iwork, liwork, info )
```

Description

This routine reorders the generalized real-Schur/Schur decomposition of a real/complex matrix pair (*A*, *B*) (in terms of an orthogonal/unitary equivalence transformation $Q' * (A, B) * Z$), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the pair (*A*, *B*). The leading columns of *Q* and *Z* form orthonormal/unitary bases of the corresponding left and right eigenspaces (deflating subspaces).
(*A*, *B*) must be in generalized real-Schur/Schur canonical form (as returned by `?gges`), that is, *A* and *B* are both upper triangular.
`?tgsen` also computes the generalized eigenvalues

ω_j = (*alphar*(j) + *alphai*(j)*i)/*beta*(j)   (for real flavors)
ω_j = *alpha*(j)/*beta*(j)   (for complex flavors)
of the reordered matrix pair (*A*, *B*).

Optionally, the routine computes the estimates of reciprocal condition numbers for eigenvalues and eigenspaces. These are
Difu$[(A_{11}, B_{11}), (A_{22}, B_{22})]$ and Difl$[(A_{11}, B_{11}), (A_{22}, B_{22})]$, that is, the separation(s) between the

matrix pairs (A_{11}, B_{11}) and (A_{22}, B_{22}) that correspond to the selected cluster and the eigenvalues outside the cluster, respectively, and norms of "projections" onto left and right eigenspaces with respect to the selected cluster in the (1,1)-block.

Input Parameters

ijob	`INTEGER`. Specifies whether condition numbers are required for the cluster of eigenvalues (*pl* and *pr*) or the deflating subspaces Difu and Difl. If *ijob* =0, only reorder with respect to *select*; If *ijob* =1, reciprocal of norms of "projections" onto left and right eigenspaces with respect to the selected cluster (*pl* and *pr*); If *ijob* =2, compute upper bounds on Difu and Difl, using F-norm-based estimate (*dif* (1:2)); If *ijob* =3, compute estimate of Difu and Difl, using 1-norm-based estimate (*dif* (1:2)). This option is about 5 times as expensive as *ijob* =2; If *ijob* =4, compute *pl*, *pr* and *dif* (i.e., options 0, 1 and 2 above). This is an economic version to get it all; If *ijob* =5, compute *pl*, *pr* and *dif* (i.e., options 0, 1 and 3 above).
wantq, wantz	`LOGICAL`. If *wantq* = `.TRUE.`, update the left transformation matrix *Q*; If *wantq* = `.FALSE.`, do not update *Q*; If *wantz* = `.TRUE.`, update the right transformation matrix *Z*; If *wantz* = `.FALSE.`, do not update *Z*.
select	`LOGICAL`. Array, `DIMENSION` at least max (1, *n*). Specifies the eigenvalues in the selected cluster. To select an eigenvalue ω_j, *select*(*j*) must be `.TRUE.` *For real flavors*: to select a complex conjugate pair of eigenvalues ω_j and ω_{j+1}(corresponding 2 by 2 diagonal block), *select*(*j*) and/or *select*(*j*+1) must be set to `.TRUE.`; the complex conjugate ω_j and ω_{j+1} must be either both included in the cluster or both excluded.
n	`INTEGER`. The order of the matrices *A* and *B* ($n \geq 0$).
a,b,q,z,work	`REAL` for `stgsen` `DOUBLE PRECISION` for `dtgsen` `COMPLEX` for `ctgsen` `DOUBLE COMPLEX` for `ztgsen`. Arrays:

`a(lda,*)` contains the matrix *A*.
For real flavors: *A* is upper quasi-triangular, with (*A*, *B*) in generalized real Schur canonical form.
For complex flavors: *A* is upper triangular, in generalized Schur canonical form.
The second dimension of *a* must be at least max(1, *n*).

`b(ldb,*)` contains the matrix *B*.
For real flavors: *B* is upper triangular, with (*A*, *B*) in generalized real Schur canonical form.
For complex flavors: *B* is upper triangular, in generalized Schur canonical form.
The second dimension of *b* must be at least max(1, *n*).

q (*ldq*,*)
If *wantq* =.TRUE., then *q* is an *n*-by-*n* matrix;
If *wantq* =.FALSE., then *q* is not referenced.
The second dimension of *q* must be at least max(1, *n*).

z (*ldz*,*)
If *wantz* =.TRUE., then *z* is an *n*-by-*n* matrix;
If *wantz* =.FALSE., then *z* is not referenced.
The second dimension of *z* must be at least max(1, *n*).

work(*lwork*) is a workspace array. If *ijob*=0, *work* is not referenced.

lda — INTEGER. The first dimension of *a*; at least max(1, *n*).

ldb — INTEGER. The first dimension of *b*; at least max(1, *n*).

ldq — INTEGER. The first dimension of *q*; *ldq* ≥ 1.
If *wantq* =.TRUE., then *ldq* ≥ max(1,*n*).

ldz — INTEGER. The first dimension of *z*; *ldz* ≥ 1.
If *wantz* =.TRUE., then *ldz* ≥ max(1,*n*).

lwork — INTEGER. The dimension of the array *work*.
For real flavors:
If *ijob* = 1, 2, or 4, *lwork* ≥ max(4*n*+16, 2*m*(*n*–*m*)).
If *ijob* = 3 or 5, *lwork* ≥ max(4*n*+16, 4*m*(*n*–*m*)).
For complex flavors:
If *ijob* = 1, 2, or 4, *lwork* ≥ max(1, 2*m*(*n*–*m*)).
If *ijob* = 3 or 5, *lwork* ≥ max(1, 4*m*(*n*–*m*)).

iwork — INTEGER. Workspace array, DIMENSION (*liwork*).
If *ijob*=0, *iwork* is not referenced.

liwork — `INTEGER`. The dimension of the array *iwork*.
For real flavors:
If *ijob* = 1, 2, or 4, *liwork* ≥ n+6.
If *ijob* = 3 or 5, *liwork* ≥ max(n+6, 2m(n–m)).
For complex flavors:
If *ijob* = 1, 2, or 4, *liwork* ≥ n+2.
If *ijob* = 3 or 5, *liwork* ≥ max(n+2, 2m(n–m)).

Output Parameters

a, b — Overwritten by the reordered matrices *A* and *B,* respectively.

alphar,alphai — `REAL` for `stgsen`;
`DOUBLE PRECISION` for `dtgsen`.
Arrays, `DIMENSION` at least max(1,n). Contain values that form generalized eigenvalues in real flavors.
See *beta*.

alpha — `COMPLEX` for `ctgsen`;
`DOUBLE COMPLEX` for `ztgsen`.
Array, `DIMENSION` at least max(1,n). Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta — `REAL` for `stgsen`
`DOUBLE PRECISION` for `dtgsen`
`COMPLEX` for `ctgsen`
`DOUBLE COMPLEX` for `ztgsen`.
Array, `DIMENSION` at least max(1,n).
For real flavors:
On exit, (*alphar*(j) + *alphai*(j)*i)/*beta*(j), j=1,...,n, will be the generalized eigenvalues.
alphar(j) + *alphai*(j)*i and *beta*(j), j=1,...,n are the diagonals of the complex Schur form (*S,T*) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (*A,B*) were further reduced to triangular form using complex unitary transformations. If *alphai*(j) is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with *alphai*(j+1) negative.
For complex flavors:
The diagonal elements of *A* and *B*, respectively, when the pair (*A,B*) has been reduced to generalized Schur form. *alpha*(i)/*beta*(i), i=1,...,n are the generalized eigenvalues.

q	If *wantq* =.TRUE., then, on exit, *Q* has been postmultiplied by the left orthogonal transformation matrix which reorder (*A*, *B*). The leading *m* columns of *Q* form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces).
z	If *wantz* =.TRUE., then, on exit, *Z* has been postmultiplied by the left orthogonal transformation matrix which reorder (*A*, *B*). The leading *m* columns of *Z* form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces).
m	INTEGER. The dimension of the specified pair of left and right eigen-spaces (deflating subspaces); $0 \le m \le n$.
pl, pr	REAL for single precision flavors; DOUBLE PRECISION for double precision flavors. If *ijob* = 1, 4, or 5, *pl* and *pr* are lower bounds on the reciprocal of the norm of "projections" onto left and right eigenspaces with respect to the selected cluster. $0 < pl, pr \le 1$. If $m = 0$ or $m = n$, $pl = pr = 1$. If *ijob* = 0, 2 or 3, *pl* and *pr* are not referenced
dif	REAL for single precision flavors; DOUBLE PRECISION for double precision flavors. Array, DIMENSION (2). If *ijob* ≥ 2, *dif*(1:2) store the estimates of Difu and Difl. If *ijob* = 2 or 4, *dif*(1:2) are F-norm-based upper bounds on Difu and Difl. If *ijob* = 3 or 5, *dif*(1:2) are 1-norm-based estimates of Difu and Difl. If *m* = 0 or *n*, *dif*(1:2) = F-norm([*A*, *B*]). If *ijob* = 0 or 1, *dif* is not referenced.
work(1)	If *ijob* is not 0 and *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
iwork(1)	If *ijob* is not 0 and *info* = 0, on exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance. Use this *liwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *-i*, the *i*th parameter had an illegal value. If *info* = 1, Reordering of (*A, B*) failed because the transformed matrix pair

(*A, B*) would be too far from generalized Schur form; the problem is very ill-conditioned. (*A, B*) may have been partially reordered. If requested, 0 is returned in `dif(*)`, `pl` and `pr`.

?tgsyl

Solves the generalized Sylvester equation.

Syntax

```
call stgsyl ( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e,
                lde, f, ldf, scale, dif, work, lwork, iwork, info )
call dtgsyl ( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e,
                lde, f, ldf, scale, dif, work, lwork, iwork, info )
call ctgsyl ( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e,
                lde, f, ldf, scale, dif, work, lwork, iwork, info )
call ztgsyl ( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e,
                lde, f, ldf, scale, dif, work, lwork, iwork, info )
```

Description

This routine solves the generalized Sylvester equation:

$$A R - L B = scale * C$$

$$D R - L E = scale * F$$

where R and L are unknown m-by-n matrices, (A, D), (B, E) and (C, F) are given matrix pairs of size m-by-m, n-by-n and m-by-n, respectively, with real/complex entries. (A, D) and (B, E) must be in generalized real-Schur/Schur canonical form, that is, A, B are upper quasi-triangular/triangular and D, E are upper triangular.

The solution (R, L) overwrites (C, F). The factor *scale*, $0 \le scale \le 1$, is an output scaling factor chosen to avoid overflow.

In matrix notation the above equation is equivalent to the following:
solve $Zx = scale* b$, where Z is defined as

$$Z = \begin{pmatrix} kron(I_n, A) & -kron(B', I_m) \\ kron(I_n, D) & -kron(E', I_m) \end{pmatrix}$$

Here I_k is the identity matrix of size k and X' is the transpose/conjugate-transpose of X. $kron(X, Y)$ is the Kronecker product between the matrices X and Y.
If *trans* = 'T'(for real flavors), or *trans* = 'C'(for complex flavors), the routine ?tgsyl solves the transposed/conjugate-transposed system
$Z' y = scale * b$, which is equivalent to solve for R and L in

$$A' R + D' L = scale * C$$

$$R B' + L E' = scale * (-F)$$

This case (*trans* = 'T' for stgsyl/dtgsyl or *trans* = 'C' for ctgsyl/ztgsyl) is used to compute an one-norm-based estimate of Dif[(A,D), (B,E)], the separation between the matrix pairs $(A,D)$and (B,E), using slacon/clacon.

If *ijob* ≥ 1, ?tgsyl computes a Frobenius norm-based estimate of Dif[(A,D), (B,E)]. That is, the reciprocal of a lower bound on the reciprocal of the smallest singular value of Z. This is a level 3 BLAS algorithm.

Input Parameters

trans — CHARACTER*1. Must be 'N', 'T', or 'C'.
If *trans* = 'N', solve the generalized Sylvester equation.
If *trans* = 'T', solve the 'transposed' system (for real flavors only).
If *trans* = 'C', solve the ' conjugate transposed' system (for complex flavors only).

ijob — INTEGER. Specifies what kind of functionality to be performed:
If *ijob* =0, solve the generalized Sylvester equation only ;
If *ijob* =1, perform the functionality of *ijob* =0 and *ijob* =3;
If *ijob* =2, perform the functionality of *ijob* =0 and *ijob* =4;
If *ijob* =3, only an estimate of Dif[(A,D), (B,E)] is computed (look ahead strategy is used);
If *ijob* =4, only an estimate of Dif[(A,D), (B,E)] is computed (?gecon on sub-systems is used).
If *trans* = 'T'or 'C', *ijob* is not referenced.

m — INTEGER.
The order of the matrices A and D, and the row dimension of the matrices C, F, R and L.

n INTEGER.
The order of the matrices *B* and *E*, and the column dimension of the matrices *C*, *F*, *R* and *L*.

a,b,c,d,e,f,work REAL for stgsyl
DOUBLE PRECISION for dtgsyl
COMPLEX for ctgsyl
DOUBLE COMPLEX for ztgsyl.

Arrays:
a(*lda*,*) contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors)
matrix *A*.
The second dimension of *a* must be at least max(1, *m*).

b(*ldb*,*) contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors)
matrix *B*.
The second dimension of *b* must be at least max(1, *n*).

c (*ldc*,*) contains the right-hand-side of the first matrix equation in the generalized Sylvester equation (as defined by *trans*)
The second dimension of *c* must be at least max(1, *n*).

d (*ldd*,*) contains the upper triangular matrix *D*.
The second dimension of *d* must be at least max(1, *m*).

e (*lde*,*) contains the upper triangular matrix *E*.
The second dimension of *e* must be at least max(1, *n*).

f (*ldf*,*) contains the right-hand-side of the second matrix equation in the generalized Sylvester equation (as defined by *trans*)
The second dimension of *f* must be at least max(1, *n*).

work(*lwork*) is a workspace array. If *ijob*=0, *work* is not referenced.

lda INTEGER. The first dimension of *a*; at least max(1, *m*).

ldb INTEGER. The first dimension of *b*; at least max(1, *n*).

ldc INTEGER. The first dimension of *c*; at least max(1, *m*).

ldd INTEGER. The first dimension of *d*; at least max(1, *m*).

lde INTEGER. The first dimension of *e*; at least max(1, *n*).

ldf INTEGER. The first dimension of *f*; at least max(1, *m*).

`lwork` INTEGER. The dimension of the array `work` . `lwork` ≥ 1. If `ijob` = 1 or 2 and `trans` = 'N', `lwork` ≥ 2*mn*.

`iwork` INTEGER. Workspace array, DIMENSION at least (*m*+*n*+6) for real flavors, and at least (*m*+*n*+2) for complex flavors.
If `ijob`=0, `iwork` is not referenced.

Output Parameters

`c` If `ijob`=0, 1, or 2, overwritten by the solution *R*.
If `ijob`=3 or 4 and `trans` = 'N', `c` holds *R*, the solution achieved during the computation of the Dif-estimate.

`f` If `ijob`=0, 1, or 2, overwritten by the solution *L*.
If `ijob`=3 or 4 and `trans` = 'N', `f` holds *L*, the solution achieved during the computation of the Dif-estimate.

`dif` REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
On exit, `dif` is the reciprocal of a lower bound of the reciprocal of the Dif-function, i.e. `dif` is an upper bound of Dif[(*A*,*D*), (*B*,*E*)] = sigma_min(*Z*), where *Z* as in (2).
If `ijob` = 0, or `trans` = 'T'(for real flavors), or `trans` = 'C'(for complex flavors), `dif` is not touched.

`scale` REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
On exit, `scale` is the scaling factor in the generalized Sylvester equation. If 0 < `scale` < 1, `c` and `f` hold the solutions *R* and *L*, respectively, to a slightly perturbed system but the input matrices *A*, *B*, *D* and *E* have not been changed. If `scale` = 0, `c` and `f` hold the solutions *R* and *L*, respectively, to the homogeneous system with *C* = *F* = 0. Normally, `scale` = 1.

`work(1)` If `ijob` is not 0 and `info` = 0, on exit, `work(1)` contains the minimum value of `lwork` required for optimum performance. Use this `lwork` for subsequent runs.

`info` INTEGER.
If `info` = 0, the execution is successful.
If `info` = `-i`, the `i`th parameter had an illegal value.
If `info` > 0, (*A*, *D*) and (*B*, *E*) have common or close eigenvalues.

?tgsna

Estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a pair of matrices in generalized real Schur canonical form.

Syntax

```
call stgsna ( job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr,
              ldvr, s, dif, mm, m, work, lwork, iwork, info )
call dtgsna ( job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr,
              ldvr, s, dif, mm, m, work, lwork, iwork, info )
call ctgsna ( job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr,
              ldvr, s, dif, mm, m, work, lwork, iwork, info )
call ztgsna ( job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr,
              ldvr, s, dif, mm, m, work, lwork, iwork, info )
```

Description

The real flavors `stgsna/dtgsna` of this routine estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B) in generalized real Schur canonical form (or of any matrix pair $(Q A Z^T, Q B Z^T)$ with orthogonal matrices Q and Z. (A, B) must be in generalized real Schur form (as returned by `sgges/dgges`), that is, A is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks. B is upper triangular.

The complex flavors `ctgsna/ztgsna` estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B). (A, B) must be in generalized Schur canonical form , that is, A and B are both upper triangular.

Input Parameters

job — `CHARACTER*1`. Specifies whether condition numbers are required for eigenvalues or eigenvectors .
Must be `'E'` or `'V'` or `'B'`.
If *job* = `'E'`, for eigenvalues only (compute *s*).
If *job* = `'V'`, for eigenvectors only (compute *dif*).
If *job* = `'B'`, for both eigenvalues and eigenvectors (compute both *s* and *dif*).

howmny	CHARACTER*1. Must be 'A' or 'S'. If *howmny* ='A', compute condition numbers for all eigenpairs. If *howmny* ='S', compute condition numbers for selected eigenpairs specified by the logical array *select*.
select	LOGICAL. Array, DIMENSION at least max (1, *n*). If *howmny* ='S' , *select* specifies the eigenpairs for which condition numbers are required. If *howmny*= 'A', *select* is not referenced. *For real flavors*: To select condition numbers for the eigenpair corresponding to a real eigenvalue ω_j, *select*(j) must be set to .TRUE.; to select condition numbers corresponding to a complex conjugate pair of eigenvalues ω_j and ω_{j+1}, either *select*(j) or *select*(j+1) must be set to .TRUE. *For complex flavors:* To select condition numbers for the corresponding j-th eigenvalue and/or eigenvector, *select*(j) must be set to .TRUE..
n	INTEGER. The order of the square matrix pair (A, B) ($n \geq 0$).
a,b,vl,vr,work	REAL for stgsna DOUBLE PRECISION for dtgsna COMPLEX for ctgsna DOUBLE COMPLEX for ztgsna. Arrays: *a*(*lda*,*) contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix A in the pair (A, B) . The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the upper triangular matrix B in the pair (A, B) . The second dimension of *b* must be at least max(1, *n*). If *job* ='E' or 'B', *vl*(*ldvl*,*) must contain left eigenvectors of (A, B), corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vl*, as returned by ?tgevc. If *job* ='V', *vl* is not referenced. The second dimension of *vl* must be at least max(1, *m*).

If *job* = 'E' or 'B',
vr(*ldvr*,*) must contain right eigenvectors of (*A*, *B*), corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vr*, as returned by ?tgevc.
If *job* = 'V', *vr* is not referenced.
The second dimension of *vr* must be at least max(1, *m*).

work(*lwork*) is a workspace array. If *job* = 'E', *work* is not referenced.

lda — INTEGER. The first dimension of *a*; at least max(1, *n*).

ldb — INTEGER. The first dimension of *b*; at least max(1, *n*).

ldvl — INTEGER. The first dimension of *vl*; $ldvl \geq 1$.
If *job* = 'E' or 'B', then $ldvl \geq \max(1,n)$.

ldvr — INTEGER. The first dimension of *vr*; $ldvr \geq 1$.
If *job* = 'E' or 'B', then $ldvr \geq \max(1,n)$.

mm — INTEGER. The number of elements in the arrays *s* and *dif* ($mm \geq m$).

lwork — INTEGER. The dimension of the array *work*.
For real flavors:
$lwork \geq n$.
If *job* = 'V' or 'B', $lwork \geq 2n(n+2)+16$.
For complex flavors:
$lwork \geq 1$.
If *job* = 'V' or 'B', $lwork \geq 2n^2$.

iwork — INTEGER. Workspace array, DIMENSION at least (n+6) for real flavors, and at least (n+2) for complex flavors.
If *ijob* = 'E', *iwork* is not referenced.

Output Parameters

s — REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Array, DIMENSION (*mm*).
If *job* = 'E' or 'B', contains the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array.
If *job* = 'V', *s* is not referenced.
For real flavors:
For a complex conjugate pair of eigenvalues two consecutive elements of *s* are

set to the same value. Thus, `s`(j), `dif`(j), and the j-th columns of `vl` and `vr` all correspond to the same eigenpair (but not in general the j-th eigenpair, unless all eigenpairs are selected).

dif

REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Array, DIMENSION (*mm*).
If *job* ='V' or 'B', contains the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array. If the eigenvalues cannot be reordered to compute *dif*(j), *dif*(j) is set to 0; this can only occur when the true value would be very small anyway.
If *job* ='E', *dif* is not referenced.
For real flavors:
For a complex eigenvector, two consecutive elements of *dif* are set to the same value.
For complex flavors:
For each eigenvalue/vector specified by *select*, *dif* stores a Frobenius norm-based estimate of Difl.

m

INTEGER. The number of elements in the arrays *s* and *dif* used to store the specified condition numbers; for each selected eigenvalue one element is used. If *howmny* ='A', *m* is set to *n*.

work(1)

work(1)If *job* is not 'E' and *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info

INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Generalized Singular Value Decomposition

This section describes LAPACK computational routines used for finding the generalized singular value decomposition (GSVD) of two matrices A and B as

$$U^H A Q = D_1 * (0 \;\; R),$$

$$V^H B Q = D_2 * (0 \;\; R),$$

where U, V, and Q are orthogonal/unitary matrices, R is a nonsingular upper triangular matrix, and D_1 , D_2 are "diagonal" matrices of the structure detailed in the routines description section.

Table 4-7 Computational Routines for Generalized Singular Value Decomposition

Routine name	Operation performed
?ggsvp	Computes the preprocessing decomposition for the generalized SVD
?tgsja	Computes the generalized SVD of two upper triangular or trapezoidal matrices

You can use routines listed in the above table as well as the driver routine ?ggsvd to find the GSVD of a pair of general rectangular matrices.

?ggsvp

Computes the preprocessing decomposition for the generalized SVD.

Syntax

```
call sggsvp ( jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb,
              k, l, u, ldu, v, ldv, q, ldq, iwork, tau, work, info )
call dggsvp ( jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb,
              k, l, u, ldu, v, ldv, q, ldq, iwork, tau, work, info )
call cggsvp (jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb,
       k, l, u, ldu, v, ldv, q, ldq, iwork, rwork, tau, work, info )
call zggsvp ( jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb,
       k, l, u, ldu, v, ldv, q, ldq, iwork, rwork, tau, work, info )
```

Description

This routine computes orthogonal matrices U, V and Q such that

$$U^H A Q = \begin{array}{c} \\ k \\ l \\ m-k-l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \\ 0 & 0 & 0 \end{pmatrix} \end{array}, \quad \text{if } m-k-l \geq 0$$

$$= \begin{array}{c} \\ k \\ m-k \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \end{pmatrix} \end{array}, \quad \text{if } m-k-l < 0$$

$$V^H B Q = \begin{array}{c} \\ l \\ p-l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & 0 & B_{13} \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

where the k-by-k matrix A_{12} and l-by-l matrix B_{13} are nonsingular upper triangular; A_{23} is l-by-l upper triangular if $m-k-l \geq 0$, otherwise A_{23} is $(m-k)$-by-l upper trapezoidal. The sum $k+l$ is equal to the effective numerical rank of the $(m+p)$-by-n matrix $(A^H, B^H)^H$.

This decomposition is the preprocessing step for computing the Generalized Singular Value Decomposition (GSVD), see subroutine ?ggsvd.

Input Parameters

jobu — `CHARACTER*1`. Must be `'U'` or `'N'`.
If *jobu* =`'U'`, orthogonal/unitary matrix U is computed.
If *jobu* =`'N'`, U is not computed.

jobv — `CHARACTER*1`. Must be `'V'` or `'N'`.
If *jobv* =`'V'`, orthogonal/unitary matrix V is computed.
If *jobv* =`'N'`, V is not computed.

jobq — `CHARACTER*1`. Must be `'Q'` or `'N'`.
If *jobq* =`'Q'`, orthogonal/unitary matrix Q is computed.
If *jobq* =`'N'`, Q is not computed.

m — `INTEGER`. The number of rows of the matrix A ($m \geq 0$).

p — `INTEGER`. The number of rows of the matrix B ($p \geq 0$).

n — `INTEGER`. The number of columns of the matrices A and B ($n \geq 0$).

a, b, tau, work — `REAL` for `sggsvp`
`DOUBLE PRECISION` for `dggsvp`
`COMPLEX` for `cggsvp`
`DOUBLE COMPLEX` for `zggsvp`.
Arrays:
a(*lda*,*) contains the m-by-n matrix A.
The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the p-by-n matrix B.
The second dimension of *b* must be at least max(1, *n*).

tau(*) is a workspace array. The dimension of *tau* must be at least max(1, *n*).

work(*) is a workspace array. The dimension of *work* must be at least max(1, 3*n*, *m*, *p*).

lda — `INTEGER`. The first dimension of *a*; at least max(1, *m*).

ldb — `INTEGER`. The first dimension of *b*; at least max(1, *p*).

tola, tolb — `REAL` for single-precision flavors
`DOUBLE PRECISION` for double-precision flavors.
tola and *tolb* are the thresholds to determine the effective numerical rank of matrix B and a subblock of A. Generally, they are set to
tola = max(*m*, *n*)*||*A*||*MACHEPS,
tolb = max(*p*, *n*)*||*B*||*MACHEPS.
The size of *tola* and *tolb* may affect the size of backward errors of the decomposition.

ldu — `INTEGER`. The first dimension of the output array *u*. *ldu* ≥ max(1, *m*) if *jobu* = `'U'`; *ldu* ≥ 1 otherwise.

ldv — `INTEGER`. The first dimension of the output array *v*. *ldv* ≥ max(1, *p*) if *jobv* = `'V'`; *ldv* ≥ 1 otherwise.

ldq — `INTEGER`. The first dimension of the output array *q*. *ldq* ≥ max(1, *n*) if *jobq* = `'Q'`; *ldq* ≥ 1 otherwise.

iwork — `INTEGER`. Workspace array, `DIMENSION` at least max(1, *n*) .

rwork — `REAL` for `cggsvp`
`DOUBLE PRECISION` for `zggsvp`.
Workspace array, `DIMENSION` at least max(1, 2*n*). Used in complex flavors only.

Output Parameters

a — Overwritten by the triangular (or trapezoidal) matrix described in the *Description* section.

b — Overwritten by the triangular matrix described in the *Description* section.

k, l — `INTEGER`.
On exit, *k* and *l* specify the dimension of subblocks.
The sum *k* + *l* is equal to effective numerical rank of $(A^H, B^H)^H$.

u, v, q — `REAL` for `sggsvp`
`DOUBLE PRECISION` for `dggsvp`
`COMPLEX` for `cggsvp`
`DOUBLE COMPLEX` for `zggsvp`.
Arrays:

If `jobu` ='U', `u(ldu,*)` contains the orthogonal/unitary matrix *U*.
The second dimension of `u` must be at least max(1, `m`).
If `jobu` ='N', `u` is not referenced.

If `jobv` ='V', `v(ldv,*)` contains the orthogonal/unitary matrix *V*.
The second dimension of `v` must be at least max(1, `m`).
If `jobv` ='N', `v` is not referenced.

If `jobq` ='Q', `q(ldq,*)` contains the orthogonal/unitary matrix *Q*.
The second dimension of `q` must be at least max(1, `n`).
If `jobq` ='N', `q` is not referenced.

`info` INTEGER.
If `info` = 0, the execution is successful.
‘If `info` = `-i`, the `i`th parameter had an illegal value.

?tgsja

Computes the generalized SVD of two upper triangular or trapezoidal matrices.

Syntax

```
call stgsja ( jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola,
      tolb, alpha, beta, u, ldu, v, ldv, q, ldq, work, ncycle, info )
call dtgsja ( jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola,
      tolb, alpha, beta, u, ldu, v, ldv, q, ldq, work, ncycle, info )
call ctgsja ( jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola,
      tolb, alpha, beta, u, ldu, v, ldv, q, ldq, work, ncycle, info )
call ztgsja ( jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola,
      tolb, alpha, beta, u, ldu, v, ldv, q, ldq, work, ncycle, info )
```

Description

This routine computes the generalized singular value decomposition (GSVD) of two real/complex upper triangular (or trapezoidal) matrices *A* and *B*. On entry, it is assumed that matrices *A* and *B* have the following forms, which may be obtained by the preprocessing subroutine ?ggsvp from a general *m*-by-*n* matrix *A* and *p*-by-*n* matrix *B*:

$$A = \begin{array}{c} \\ k \\ l \\ m-k-l \end{array}\begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \\ 0 & 0 & 0 \end{pmatrix} \end{array}, \quad \text{if } m-k-l \ge 0$$

$$= \begin{array}{c} \\ k \\ m-k \end{array}\begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \end{pmatrix} \end{array}, \quad \text{if } m-k-l < 0$$

$$B = \begin{array}{c} \\ l \\ p-l \end{array}\begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & 0 & B_{13} \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

where the k-by-k matrix A_{12} and l-by-l matrix B_{13} are nonsingular upper triangular; A_{23} is l-by-l upper triangular if $m-k-l \geq 0$, otherwise A_{23} is $(m-k)$-by-l upper trapezoidal.

On exit,

$U^H A Q = D_1*(0 \quad R)$, $V^H B Q = D_2*(0 \quad R)$,
where U, V and Q are orthogonal/unitary matrices, R is a nonsingular upper triangular matrix, and D_1 and D_2 are "diagonal" matrices, which are of the following structures:

If $m-k-l \geq 0$,

$$D_1 = \begin{array}{c} \\ k \\ l \\ m-k-l \end{array}\begin{array}{c} \begin{array}{cc} k & l \end{array} \\ \begin{pmatrix} I & 0 \\ 0 & C \\ 0 & 0 \end{pmatrix} \end{array}$$

$$D_2 = \begin{array}{c} \\ l \\ p-l \end{array}\begin{array}{c} \begin{array}{cc} k & l \end{array} \\ \begin{pmatrix} 0 & S \\ 0 & 0 \end{pmatrix} \end{array}$$

$$(0 \quad R) = \begin{array}{c} \\ k \\ l \end{array}\begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & R_{11} & R_{12} \\ 0 & 0 & R_{22} \end{pmatrix} \end{array}$$

where

C = diag ($alpha(k+1),...,alpha(k+l)$)
S = diag ($beta(k+1),...,beta(k+l)$)
$C^2 + S^2 = I$
R is stored in $a(1:k+l, n-k-l+1:n)$ on exit.

If $m-k-l < 0$,

$$D_1 = \begin{array}{c} \\ k \\ m-k \end{array} \begin{array}{c} \begin{array}{ccc} k & m-k & k+l-m \end{array} \\ \begin{pmatrix} I & 0 & 0 \\ 0 & C & 0 \end{pmatrix} \end{array}$$

$$D_2 = \begin{array}{c} \\ m-k \\ k+l-m \\ p-l \end{array} \begin{array}{c} \begin{array}{ccc} k & m-k & k+l-m \end{array} \\ \begin{pmatrix} 0 & S & 0 \\ 0 & 0 & I \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$\begin{pmatrix} 0 & R \end{pmatrix} = \begin{array}{c} \\ k \\ m-k \\ k+l-m \end{array} \begin{array}{c} \begin{array}{cccc} n-k-l & k & m-k & k+l-m \end{array} \\ \begin{pmatrix} 0 & R_{11} & R_{12} & R_{13} \\ 0 & 0 & R_{22} & R_{23} \\ 0 & 0 & 0 & R_{33} \end{pmatrix} \end{array}$$

where

C = diag ($alpha(k+1),...,alpha(m)$),
S = diag ($beta(k+1),...,beta(m)$),
$C^2 + S^2 = I$

On exit, $\begin{pmatrix} R_{11} R_{12} R_{13} \\ 0 \ R_{22} R_{23} \end{pmatrix}$ is stored in $a(1:m, n-k-l+1:n)$ and R_{33} is stored in $b(m-k+1:l, n+m-k-l+1:n)$.

The computation of the orthogonal/unitary transformation matrices *U, V* or *Q* is optional. These matrices may either be formed explicitly, or they may be postmultiplied into input matrices U_1, V_1, or Q_1.

Input Parameters

jobu — `CHARACTER*1`. Must be `'U'`, `'I'`, or `'N'`.
If *jobu* = `'U'`, *u* must contain an orthogonal/unitary matrix U_1 on entry.
If *jobu* = `'I'`, *u* is initialized to the unit matrix.
If *jobu* = `'N'`, *u* is not computed.

jobv — `CHARACTER*1`. Must be `'V'`, `'I'`, or `'N'`.
If *jobv* = `'V'`, *v* must contain an orthogonal/unitary matrix V_1 on entry.
If *jobv* = `'I'`, *v* is initialized to the unit matrix.
If *jobv* = `'N'`, *v* is not computed.

jobq — `CHARACTER*1`. Must be `'Q'`, `'I'`, or `'N'`.
If *jobq* = `'Q'`, *q* must contain an orthogonal/unitary matrix Q_1 on entry.
If *jobq* = `'I'`, *q* is initialized to the unit matrix.
If *jobq* = `'N'`, *q* is not computed.

m — `INTEGER`. The number of rows of the matrix *A* ($m \ge 0$).

p — `INTEGER`. The number of rows of the matrix *B* ($p \ge 0$).

n — `INTEGER`. The number of columns of the matrices *A* and *B* ($n \ge 0$).

k, l — `INTEGER`. Specify the subblocks in the input matrices *A* and *B*, whose GSVD is going to be computed by `?tgsja`.

a, b, u, v, q, work — `REAL` for `stgsja`
`DOUBLE PRECISION` for `dtgsja`
`COMPLEX` for `ctgsja`
`DOUBLE COMPLEX` for `ztgsja`.
Arrays:
a(*lda*,*) contains the *m*-by-*n* matrix *A*.
The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the *p*-by-*n* matrix *B*.
The second dimension of *b* must be at least max(1, *n*).

If *jobu* = `'U'`, *u*(*ldu*,*) must contain a matrix U_1 (usually the orthogonal/unitary matrix returned by `?ggsvp`).
The second dimension of *u* must be at least max(1, *m*).

If *jobv* = `'V'`, *v*(*ldv*,*) must contain a matrix V_1 (usually the orthogonal/unitary matrix returned by `?ggsvp`).
The second dimension of *v* must be at least max(1, *p*).

If *jobq* ='Q', *q*(*ldq*, *) must contain a matrix Q_1 (usually the orthogonal/unitary matrix returned by ?ggsvp). The second dimension of *q* must be at least max(1, *n*).

work(*) is a workspace array. The dimension of *work* must be at least max(1, 2*n*).

lda — INTEGER. The first dimension of *a*; at least max(1, *m*).

ldb — INTEGER. The first dimension of *b*; at least max(1, *p*).

ldu — INTEGER. The first dimension of the array *u*.
ldu ≥ max(1, *m*) if *jobu* ='U'; *ldu* ≥ 1 otherwise.

ldv — INTEGER. The first dimension of the array *v*.
ldv ≥ max(1, *p*) if *jobv* ='V'; *ldv* ≥ 1 otherwise.

ldq — INTEGER. The first dimension of the array *q*.
ldq ≥ max(1, *n*) if *jobq* ='Q'; *ldq* ≥ 1 otherwise.

tola, tolb — REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
tola and *tolb* are the convergence criteria for the Jacobi-Kogbetliantz iteration procedure. Generally, they are the same as used in ?ggsvp :
tola = max(*m*, *n*)*||*A*||*MACHEPS,
tolb = max(*p*, *n*)*||*B*||*MACHEPS.

Output Parameters

a — On exit, *a*(*n*-*k*+1:*n*, 1:min(*k*+*l*, *m*)) contains the triangular matrix *R* or part of *R*.

b — On exit, if necessary, *b*(*m*-*k*+1: *l*, *n*+*m*-*k*-*l*+1: *n*)) contains a part of *R*.

alpha, beta — REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Arrays, DIMENSION at least max(1,*n*).
Contain the generalized singular value pairs of *A* and *B*:

alpha(1:*k*) = 1,
beta(1:*k*) = 0,

and if *m*-*k*-*l* ≥ 0,
alpha(*k*+1:*k*+*l*) = diag(*C*),
beta(*k*+1:*k*+*l*) = diag(*S*),

or if $m-k-l < 0$,
alpha(k+1:m)= C, *alpha*(m+1:k+l)= 0
beta(k+1:m) = S, *beta*(m+1:k+l) = 1.

Furthermore, if $k+l < n$,
alpha(k+l+1:n) = 0 and
beta(k+l+1:n) = 0.

u — If *jobu* ='I', *u* contains the orthogonal/unitary matrix U.
If *jobu* ='U', *u* contains the product U_1U.
If *jobu* ='N', *u* is not referenced.

v — If *jobv* ='I', *v* contains the orthogonal/unitary matrix U.
If *jobv* ='V', *v* contains the product V_1V.
If *jobv* ='N', *v* is not referenced.

q — If *jobq* ='I', *q* contains the orthogonal/unitary matrix U.
If *jobq* ='Q', *q* contains the product Q_1Q.
If *jobq* ='N', *q* is not referenced.

ncycle — INTEGER. The number of cycles required for convergence.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.
If *info* = 1, the procedure does not converge after MAXIT cycles.

Driver Routines

Each of the LAPACK driver routines solves a complete problem.
To arrive at the solution, driver routines typically call a sequence of appropriate computational routines.
Driver routines are described in the following sections:

Linear Least Squares (LLS) Problems
Generalized LLS Problems
Symmetric Eigenproblems
Nonsymmetric Eigenproblems
Singular Value Decomposition
Generalized Symmetric Definite Eigenproblems
Generalized Nonsymmetric Eigenproblems

Linear Least Squares (LLS) Problems

This section describes LAPACK driver routines used for solving linear least-squares problems. Table 4-8 lists routines described in more detail below.

Table 4-8 Driver Routines for Solving LLS Problems

Routine Name	Operation performed
?gels	Uses QR or LQ factorization to solve a overdetermined or underdetermined linear system with full rank matrix.
?gelsy	Computes the minimum-norm solution to a linear least squares problem using a complete orthogonal factorization of A.
?gelss	Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A.
?gelsd	Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A and a divide and conquer method.

?gels

Uses QR or LQ factorization to solve a overdetermined or underdetermined linear system with full rank matrix.

Syntax

```
call sgels ( trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info )
call dgels ( trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info )
call cgels ( trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info )
call zgels ( trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info )
```

Description

This routine solves overdetermined or underdetermined real/ complex linear systems involving an `m`-by-`n` matrix *A*, or its transpose/ conjugate-transpose, using a *QR* or *LQ* factorization of *A*. It is assumed that *A* has full rank.

The following options are provided:

1. If *trans* = 'N' and $m \geq n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

$$\text{minimize} \quad \| b - A x \|_2$$

2. If *trans* = 'N' and $m < n$: find the minimum norm solution of an underdetermined system $A X = B$.

3. If *trans* = 'T' or 'C' and $m \geq n$: find the minimum norm solution of an undetermined system $A^H X = B$.

4. If *trans* = 'T' or 'C' and $m < n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

$$\text{minimize} \quad \| b - A^H x \|_2$$

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-`nrh` solution matrix *X*.

Input Parameters

trans — CHARACTER*1. Must be 'N', 'T', or 'C'.
If *trans* = 'N', the linear system involves matrix A;
If *trans* = 'T', the linear system involves the transposed matrix A^T (for real flavors only);
If *trans* = 'C', the linear system involves the conjugate-transposed matrix A^H (for complex flavors only).

m — INTEGER. The number of rows of the matrix A ($m \geq 0$).

n — INTEGER. The number of columns of the matrix A ($n \geq 0$).

nrhs — INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).

a, b, work — REAL for sgels
DOUBLE PRECISION for dgels
COMPLEX for cgels
DOUBLE COMPLEX for zgels.
Arrays:
a(*lda*,*) contains the *m*-by-*n* matrix A.
The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the matrix B of right hand side vectors, stored columnwise; B is *m*-by-*nrhs* if *trans* = 'N', or *n*-by-*nrhs* if *trans* = 'T'or 'C'.
The second dimension of *b* must be at least max(1, *nrhs*).

work(*lwork*) is a workspace array.

lda — INTEGER. The first dimension of *a*; at least max(1, *m*).

ldb — INTEGER. The first dimension of *b*; must be at least max(1, *m*, *n*).

lwork — INTEGER. The size of the *work* array; must be at least min (*m*, *n*) +max(1, *m*, *n*, *nrhs*).
See *Application notes* for the suggested value of *lwork*.

Output Parameters

a — On exit, overwritten by the factorization data as follows:

if $m \geq n$, array *a* contains the details of the *QR* factorization of the matrix *A* as returned by ?geqrf;
if $m < n$, array *a* contains the details of the *LQ* factorization of the matrix *A* as returned by ?gelqf.

b Overwritten by the solution vectors, stored columnwise: If *trans* = 'N' and $m \geq n$, rows 1 to *n* of *b* contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements n+1 to *m* in that column;
If *trans* = 'N' and $m < n$, rows 1 to *n* of *b* contain the minimum norm solution vectors;
if *trans* = 'T' or 'C' and $m \geq n$, rows 1 to *m* of *b* contain the minimum norm solution vectors;
if *trans* = 'T' or 'C' and $m < n$, rows 1 to *m* of *b* contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements m+1 to *n* in that column.

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For better performance, try using
lwork =min (*m*, *n*) +max(1, *m*, *n*, *nrhs*)* *blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

?gelsy

Computes the minimum-norm solution to a linear least squares problem using a complete orthogonal factorization of A.

Syntax

```
call sgelsy ( m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work,
              lwork, info )
call dgelsy ( m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work,
              lwork, info )
call cgelsy ( m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work,
              lwork, rwork, info )
call zgelsy ( m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work,
              lwork, rwork, info )
```

Description

This routine computes the minimum-norm solution to a real/complex linear least squares problem:

$$\text{minimize} \quad \| b - A x \|_2$$

using a complete orthogonal factorization of A. A is an `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.

The routine first computes a QR factorization with column pivoting:

$$AP = Q\begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix}$$

with R_{11} defined as the largest leading submatrix whose estimated condition number is less than 1/`rcond`. The order of R_{11}, `rank`, is the effective rank of A.
Then, R_{22} is considered to be negligible, and R_{12} is annihilated by orthogonal/unitary transformations from the right, arriving at the complete orthogonal factorization:

$$AP = Q\begin{pmatrix} T_{11} & 0 \\ 0 & 0 \end{pmatrix} Z$$

The minimum-norm solution is then

$$x = PZ^H \begin{pmatrix} T_{11}^{-1} Q_1^H b \\ 0 \end{pmatrix}$$

where Q_1 consists of the first `rank` columns of Q. This routine is basically identical to the original `?gelsx` except three differences:

- The call to the subroutine `?geqpf` has been substituted by the call to the subroutine `?geqp3`. This subroutine is a BLAS-3 version of the *QR* factorization with column pivoting.
- Matrix *B* (the right hand side) is updated with BLAS-3.
- The permutation of matrix *B* (the right hand side) is faster and more simple.

Input Parameters

m	`INTEGER`. The number of rows of the matrix *A* ($m \geq 0$).
n	`INTEGER`. The number of columns of the matrix *A* ($n \geq 0$).
nrhs	`INTEGER`. The number of right-hand sides; the number of columns in *B* ($nrhs \geq 0$).
a, b, work	`REAL` for `sgelsy` `DOUBLE PRECISION` for `dgelsy` `COMPLEX` for `cgelsy` `DOUBLE COMPLEX` for `zgelsy`. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the *m*-by-*nrhs* right hand side matrix *B*. The second dimension of *b* must be at least max(1, *nrhs*). *work*(*lwork*) is a workspace array.

lda	`INTEGER`. The first dimension of *a*; at least max(1, *m*).
ldb	`INTEGER`. The first dimension of *b*; must be at least max(1, *m*, *n*).
jpvt	`INTEGER`. Array, `DIMENSION` at least max(1, *n*). On entry, if *jpvt*(*i*)≠ 0, the *i*th column of *A* is permuted to the front of *AP*, otherwise the *i*th column of *A* is a free column.
rcond	`REAL` for single-precision flavors `DOUBLE PRECISION` for double-precision flavors. *rcond* is used to determine the effective rank of *A*, which is defined as the order of the largest leading triangular submatrix R_{11} in the *QR* factorization with pivoting of *A*, whose estimated condition number < 1/*rcond*.
lwork	`INTEGER`. The size of the *work* array. See *Application notes* for the suggested value of *lwork*.
rwork	`REAL` for `cgelsy` `DOUBLE PRECISION` for `zgelsy`. Workspace array, `DIMENSION` at least max(1, 2*n*). Used in complex flavors only.

Output Parameters

a	On exit, overwritten by the details of the complete orthogonal factorization of *A*.
b	Overwritten by the *n*-by-*nrhs* solution matrix *X*.
jpvt	On exit, if *jpvt*(*i*)= *k*, then the *i*th column of *AP* was the *k*th column of *A*.
rank	`INTEGER`. The effective rank of *A*, that is, the order of the submatrix R_{11}. This is the same as the order of the submatrix T_{11} in the complete orthogonal factorization of *A*.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For real flavors:

The unblocked strategy requires that:

$lwork \geq \max(mn+3n+1, 2*mn + nrhs)$,

where $mn = \min(m, n)$.

The block algorithm requires that:

$lwork \geq \max(mn+2n+nb*(n+1), 2*mn+nb*nrhs)$,

where *nb* is an upper bound on the blocksize returned by `ilaenv` for the routines `sgeqp3/dgeqp3`, `stzrzf/dtzrzf`, `stzrqf/dtzrqf`, `sormqr/dormqr`, and `sormrz/dormrz`.

For complex flavors:

The unblocked strategy requires that:

$lwork \geq mn + \max(2*mn, n+1, mn + nrhs)$,

where $mn = \min(m, n)$.

The block algorithm requires that:

$lwork \geq mn + \max(2*mn, nb*(n+1), mn+mn*nb, mn+nb*nrhs)$,

where *nb* is an upper bound on the blocksize returned by `ilaenv` for the routines `cgeqp3/zgeqp3`, `ctzrzf/ztzrzf`, `ctzrqf/ztzrqf`, `cunmqr/zunmqr`, and `cunmrz/zunmrz`.

?gelss

Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A.

Syntax

```
call sgelss ( m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work,
              lwork, info )
call dgelss ( m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work,
              lwork, info )
call cgelss ( m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work,
              lwork, rwork, info )
call zgelss ( m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work,
              lwork, rwork, info )
```

Description

This routine computes the minimum norm solution to a real linear least squares problem:

$$\text{minimize} \quad \| b - A x \|_2$$

using the singular value decomposition (SVD) of A. A is an *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.

The effective rank of A is determined by treating as zero those singular values which are less than *rcond* times the largest singular value.

Input Parameters

m	INTEGER. The number of rows of the matrix A ($m \geq 0$).
n	INTEGER. The number of columns of the matrix A ($n \geq 0$).
nrhs	INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).

a, b, work	`REAL` for `sgelss` `DOUBLE PRECISION` for `dgelss` `COMPLEX` for `cgelss` `DOUBLE COMPLEX` for `zgelss`. Arrays: `a(lda,*)` contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). `b(ldb,*)` contains the *m*-by-*nrhs* right hand side matrix *B* . The second dimension of *b* must be at least max(1, *nrhs*). `work(lwork)` is a workspace array.
lda	`INTEGER`. The first dimension of *a*; at least max(1, *m*).
ldb	`INTEGER`. The first dimension of *b*; must be at least max(1, *m*, *n*).
rcond	`REAL` for single-precision flavors `DOUBLE PRECISION` for double-precision flavors. *rcond* is used to determine the effective rank of *A*. Singular values $s(i) \le$ *rcond* $*s(1)$ are treated as zero. If *rcond* < 0, machine precision is used instead.
lwork	`INTEGER`. The size of the *work* array; *lwork* ≥ 1. See *Application notes* for the suggested value of *lwork*.
rwork	`REAL` for `cgelss` `DOUBLE PRECISION` for `zgelss`. Workspace array used in complex flavors only. `DIMENSION` at least max(1, 5*min(*m*, *n*)).

Output Parameters

a	On exit, the first min(*m*, *n*) rows of *A* are overwritten with its right singular vectors, stored row-wise.
b	Overwritten by the *n*-by-*nrhs* solution matrix *X*. If $m \ge n$ and *rank* = *n*, the residual sum-of-squares for the solution in the *i*-th column is given by the sum of squares of elements *n*+1:*m* in that column.

s REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Array, DIMENSION at least max(1, min(*m*, *n*)). The singular values of *A* in decreasing order. The condition number of *A* in the 2-norm is

$$k_2(A) = s(1) / s(\min(m, n)) .$$

rank INTEGER.
The effective rank of *A*, that is, the number of singular values which are greater than *rcond* **s*(1).

work(1) If *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, then the algorithm for computing the SVD failed to converge; *i* indicates the number of off-diagonal elements of an intermediate bidiagonal form which did not converge to zero.

Application Notes

For real flavors:

$$lwork \geq 3*\min(m, n) + \max(2*\min(m, n), \max(m, n), nrhs)$$

For complex flavors:

$$lwork \geq 2*\min(m, n) + \max(m, n, nrhs)$$

For good performance, *lwork* should generally be larger. If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

?gelsd

Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A and a divide and conquer method.

Syntax

```
call sgelsd ( m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work,
              lwork, iwork, info )
call dgelsd ( m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work,
              lwork, iwork, info )
call cgelsd ( m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work,
              lwork, rwork, iwork, info )
call zgelsd ( m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work,
              lwork, rwork, iwork, info )
```

Description

This routine computes the minimum-norm solution to a real linear least squares problem:

$$\text{minimize} \quad \| b - A x \|_2$$

using the singular value decomposition (SVD) of A. A is an `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.

The problem is solved in three steps:

1. Reduce the coefficient matrix A to bidiagonal form with Householder transformations, reducing the original problem into a "bidiagonal least squares problem" (BLS).
2. Solve the BLS using a divide and conquer approach.
3. Apply back all the Householder transformations to solve the original least squares problem.

The effective rank of A is determined by treating as zero those singular values which are less than `rcond` times the largest singular value.

Input Parameters

m	INTEGER. The number of rows of the matrix *A* ($m \geq 0$).
n	INTEGER. The number of columns of the matrix *A* ($n \geq 0$).
nrhs	INTEGER. The number of right-hand sides; the number of columns in *B* ($nrhs \geq 0$).
a, b, work	REAL for sgelsd DOUBLE PRECISION for dgelsd COMPLEX for cgelsd DOUBLE COMPLEX for zgelsd. Arrays: a(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of a must be at least max(1, *n*). b(*ldb*,*) contains the *m*-by-*nrhs* right hand side matrix *B* . The second dimension of b must be at least max(1, *nrhs*). *work*(*lwork*) is a workspace array.
lda	INTEGER. The first dimension of a; at least max(1, *m*).
ldb	INTEGER. The first dimension of b; must be at least max(1, *m*, *n*).
rcond	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. *rcond* is used to determine the effective rank of *A*. Singular values $s(i) \leq rcond * s(1)$ are treated as zero. If *rcond* < 0, machine precision is used instead.
lwork	INTEGER. The size of the *work* array; *lwork* ≥ 1. See *Application notes* for the suggested value of *lwork*.
iwork	INTEGER. Workspace array. See *Application notes* for the suggested dimension of *iwork*.
rwork	REAL for cgelsd DOUBLE PRECISION for zgelsd . Workspace array, used in complex flavors only. See *Application notes* for the suggested dimension of *rwork*.

Output Parameters

a — On exit, *A* has been overwritten.

b — Overwritten by the *n*-by-*nrhs* solution matrix *X*.

If $m \ge n$ and $rank = n$, the residual sum-of-squares for the solution in the *i*-th column is given by the sum of squares of elements $n+1{:}m$ in that column.

s — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Array, DIMENSION at least max(1, min(*m*, *n*)). The singular values of *A* in decreasing order. The condition number of *A* in the 2-norm is

$$k_2(A) = s(1) / s(\min(m, n)) .$$

rank — INTEGER.
The effective rank of *A*, that is, the number of singular values which are greater than *rcond* **s*(1).

work(1) — If *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, then the algorithm for computing the SVD failed to converge; *i* indicates the number of off-diagonal elements of an intermediate bidiagonal form which did not converge to zero.

Application Notes

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

The exact minimum amount of workspace needed depends on *m*, *n* and *nrhs*. The size *lwork* of the workspace array *work* must be as given below.

For real flavors:

If $m \ge n$,
$lwork \ge 12n + 2n*smlsiz + 8n*nlvl + n*nrhs + (smlsiz+1)^2$;

If $m < n$,
$lwork \ge 12m + 2m*smlsiz + 8m*nlvl + m*nrhs + (smlsiz+1)^2$;

For complex flavors:

If $m \ge n$,
`lwork` $\ge 2n + n*nrhs$;

If $m < n$,
`lwork` $\ge 2m + m*nrhs$;

where *smlsiz* is returned by `ilaenv` and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25), and
$nlvl = \mathrm{INT}(\log_2(\min(m, n)/(smlsiz+1))) + 1$.

For good performance, `lwork` should generally be larger. If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The dimension of the workspace array `iwork` must be at least
$3*\min(m, n)*nlvl + 11*\min(m, n)$.

The dimension `lrwork` of the workspace array `rwork` (for complex flavors) must be at least:
If $m \ge n$,
`lrwork` $\ge 10n + 2n*smlsiz + 8n*nlvl + 3*smlsiz*nrhs + (smlsiz+1)^2$;

If $m < n$,
`lrwork` $\ge 10m + 2m*smlsiz + 8m*nlvl + 3*smlsiz*nrhs + (smlsiz+1)^2$.

Generalized LLS Problems

This section describes LAPACK driver routines used for solving generalized linear least-squares problems. Table 4-9 lists routines described in more detail below.

Table 4-9 Driver Routines for Solving Generalized LLS Problems

Routine Name	Operation performed
?gglse	Solves the linear equality-constrained least squares problem using a generalized RQ factorization.
?ggglm	Solves a general Gauss-Markov linear model problem using a generalized QR factorization.

?gglse

Solves the linear equality-constrained least squares problem using a generalized RQ factorization.

Syntax

```
call sgglse ( m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info )
call dgglse ( m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info )
call cgglse ( m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info )
call zgglse ( m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info )
```

Description

This routine solves the linear equality-constrained least squares (LSE) problem:

$$\text{minimize} \quad \| c - A x \|_2 \quad \text{subject to} \quad B x = d$$

where A is an m-by-n matrix, B is a p-by-n matrix, c is a given m-vector, and d is a given p-vector. It is assumed that $p \le n \le m+p$, and

$$\text{rank}(B) = p \quad \text{and} \quad \text{rank}\begin{pmatrix} A \\ B \end{pmatrix} = n \quad .$$

These conditions ensure that the LSE problem has a unique solution, which is obtained using a generalized *RQ* factorization of the matrices *B* and *A*.

Input Parameters

m — `INTEGER`. The number of rows of the matrix A ($m \ge 0$).

n — `INTEGER`. The number of columns of the matrices A and B ($n \ge 0$).

p — `INTEGER`. The number of rows of the matrix B ($0 \le p \le n \le m+p$).

a, b, c, d, work — `REAL` for `sgglse`
`DOUBLE PRECISION` for `dgglse`
`COMPLEX` for `cgglse`
`DOUBLE COMPLEX` for `zgglse`.

Arrays:
a(*lda*,*) contains the m-by-n matrix A.
The second dimension of *a* must be at least max(1, n).

b(*ldb*,*) contains the p-by-n matrix B.
The second dimension of *b* must be at least max(1, n).

c(*), dimension at least max(1, m), contains the right hand side vector for the least squares part of the LSE problem.
d(*), dimension at least max(1, p), contains the right hand side vector for the constrained equation.
work(*lwork*) is a workspace array.

lda — `INTEGER`. The first dimension of *a*; at least max(1, m).

ldb — `INTEGER`. The first dimension of *b*; at least max(1, p).

lwork — `INTEGER`. The size of the *work* array;
lwork $\ge$ max(1, $m+n+p$). See *Application notes* for the suggested value of *lwork*.

Output Parameters

x — `REAL` for `sgglse`
`DOUBLE PRECISION` for `dgglse`
`COMPLEX` for `cgglse`
`DOUBLE COMPLEX` for `zgglse`.
Array, `DIMENSION` at least max(1, n).
On exit, contains the solution of the LSE problem.

a, b, d — On exit, these arrays are overwritten.

c	On exit, the residual sum-of-squares for the solution is given by the sum of squares of elements *n*-*p*+1 to *m* of vector *c*.
work(1)	If *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For optimum performance use

$$lwork \geq p + \min(m, n) + \max(m, n) * nb,$$

where *nb* is an upper bound for the optimal blocksizes for ?geqrf, ?gerqf, ?ormqr/?unmqr and ?ormrq/?unmrq.

?ggglm

Solves a general Gauss-Markov linear model problem using a generalized QR factorization.

Syntax

```
call sggglm ( n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info )
call dggglm ( n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info )
call cggglm ( n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info )
call zggglm ( n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info )
```

Description

This routine solves a general Gauss-Markov linear model (GLM) problem:

$$\text{minimize}_x \ \|y\|_2 \quad \text{subject to} \quad d = Ax + By$$

where A is an n-by-m matrix, B is an n-by-p matrix, and d is a given n-vector.
It is assumed that $m \le n \le m+p$, and

$$\text{rank}(A) = m \quad \text{and} \quad \text{rank}(\,A\ B\,) = n\ .$$

Under these assumptions, the constrained equation is always consistent, and there is a unique solution x and a minimal 2-norm solution y, which is obtained using a generalized QR factorization of A and B.
In particular, if matrix B is square nonsingular, then the problem GLM is equivalent to the following weighted linear least squares problem

$$\text{minimize}_x \ \|B^{-1}(d - Ax)\|_2\ .$$

Input Parameters

n — `INTEGER`. The number of rows of the matrices A and B ($n \ge 0$).

m — `INTEGER`. The number of columns in A ($m \ge 0$).

p — `INTEGER`. The number of columns in B ($p \ge n - m$).

a, b, d, work — `REAL` for `sggglm`
`DOUBLE PRECISION` for `dggglm`
`COMPLEX` for `cggglm`
`DOUBLE COMPLEX` for `zggglm`.

Arrays:
`a(lda,*)` contains the n-by-m matrix A.
The second dimension of `a` must be at least max(1, m).

`b(ldb,*)` contains the *n*-by-*p* matrix *B*.
The second dimension of *b* must be at least max(1, *p*).

`d(*)`, dimension at least max(1, *n*), contains the left hand side of the GLM equation.
`work(lwork)` is a workspace array.

lda	`INTEGER`. The first dimension of *a*; at least max(1, *n*).
ldb	`INTEGER`. The first dimension of *b*; at least max(1, *n*).
lwork	`INTEGER`. The size of the *work* array; *lwork* ≥ max(1, *n*+*m*+*p*). See *Application notes* for the suggested value of *lwork*.

Output Parameters

x, y	`REAL` for `sggglm` `DOUBLE PRECISION` for `dggglm` `COMPLEX` for `cggglm` `DOUBLE COMPLEX` for `zggglm`. Arrays *x*(*), *y*(*). `DIMENSION` at least max(1, *m*) for *x* and at least max(1, *p*) for *y*. On exit, *x* and *y* are the solutions of the GLM problem.
a,b,d	On exit, these arrays are overwritten.
work(1)	If *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

For optimum performance use

lwork ≥ *m*+min(*n*, *p*)+max(*n*, *p*)**nb*,

where *nb* is an upper bound for the optimal blocksizes for `?geqrf`, `?gerqf`, `?ormqr/?unmqr` and `?ormrq/?unmrq`.

Symmetric Eigenproblems

This section describes LAPACK driver routines used for solving symmetric eigenvalue problems. See also computational routines that can be called to solve these problems. Table 4-10 lists routines described in more detail below.

Table 4-10 Driver Routines for Solving Symmetric Eigenproblems

Routine Name	Operation performed
?syev/?heev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix.
?syevd/?heevd	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian matrix using divide and conquer algorithm.
?syevx/?heevx	Computes selected eigenvalues and, optionally, eigenvectors of a symmetric / Hermitian matrix.
?syevr/?heevr	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix using the Relatively Robust Representations.
?spev/?hpev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix in packed storage.
?spevd/?hpevd	Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian matrix held in packed storage.
?spevx/?hpevx	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix in packed storage.
?sbev /?hbev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian band matrix.
?sbevd/?hbevd	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian band matrix using divide and conquer algorithm.
?sbevx/?hbevx	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian band matrix.
?stev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix.
?stevd	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric tridiagonal matrix using divide and conquer algorithm.
?stevx	Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.
?stevr	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix using the Relatively Robust Representations.

?syev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix.

Syntax

```
call ssyev ( jobz, uplo, n, a, lda, w, work, lwork, info )
call dsyev ( jobz, uplo, n, a, lda, w, work, lwork, info )
```

Description

This routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A*.

Input Parameters

jobz	`CHARACTER*1`. Must be `'N'` or `'V'`. If *jobz* =`'N'`, then only eigenvalues are computed. If *jobz* =`'V'`, then eigenvalues and eigenvectors are computed.
uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. If *uplo* = `'U'`, *a* stores the upper triangular part of *A*. If *uplo* = `'L'`, *a* stores the lower triangular part of *A*.
n	`INTEGER`. The order of the matrix *A* ($n \geq 0$).
a, work	`REAL` for `ssyev` `DOUBLE PRECISION` for `dsyev` Arrays: *a*(*lda*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A*, as specified by *uplo*. The second dimension of *a* must be at least max(1, *n*). *work*(*lwork*) is a workspace array.
lda	`INTEGER`. The first dimension of the array *a*. Must be at least max(1, *n*) .
lwork	`INTEGER`. The dimension of the array *work*. Constraint: *lwork* ≥ max(1, 3*n*-1). See *Application notes* for the suggested value of *lwork*.

Output Parameters

a — On exit, if *jobz* ='V', then if *info* = 0, array *a* contains the orthonormal eigenvectors of the matrix *A*.
If *jobz* ='N', then on exit the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

w — REAL for ssyev
DOUBLE PRECISION for dsyev
Array, DIMENSION at least max(1, *n*) .
If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order.

work(1) — On exit, if *lwork* > 0, then *work(1)* returns the required minimal size of *lwork*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Application Notes

For optimum performance use

$$lwork \geq (nb+2)*n,$$

where *nb* is the blocksize for ?sytrd returned by ilaenv.
If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

?heev

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

```
call cheev ( jobz, uplo, n, a, lda, w, work, lwork, rwork, info )
call zheev ( jobz, uplo, n, a, lda, w, work, lwork, rwork, info )
```

Description

This routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A*.

Input Parameters

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then only eigenvalues are computed.
If *jobz* = `'V'`, then eigenvalues and eigenvectors are computed.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, *a* stores the upper triangular part of *A*.
If *uplo* = `'L'`, *a* stores the lower triangular part of *A*.

n — `INTEGER`. The order of the matrix *A* ($n \geq 0$).

a, work — `COMPLEX` for `cheev`
`DOUBLE COMPLEX` for `zheev`
Arrays:
a(*lda*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A*, as specified by *uplo*.
The second dimension of *a* must be at least max(1, *n*).

work(*lwork*) is a workspace array.

lda — `INTEGER`. The first dimension of the array *a*.
Must be at least max(1, *n*) .

lwork — `INTEGER`. The dimension of the array *work*.
Constraint: *lwork* ≥ max(1, 2*n*-1). See *Application notes* for the suggested value of *lwork*.

rwork REAL for cheev
DOUBLE PRECISION for zheev .
Workspace array, DIMENSION at least max(1, 3*n*-2).

Output Parameters

a On exit, if *jobz* ='V', then if *info* = 0, array *a* contains the orthonormal eigenvectors of the matrix *A*.
If *jobz* ='N', then on exit the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

w REAL for cheev
DOUBLE PRECISION for zheev
Array, DIMENSION at least max(1, *n*) .
If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order.

work(1) On exit, if *lwork* > 0, then *work(1)* returns the required minimal size of *lwork*.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Application Notes

For optimum performance use

lwork ≥ (*nb*+1)**n*,

where *nb* is the blocksize for ?hetrd returned by ilaenv.
If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

?syevd

Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric matrix using divide and conquer algorithm.

Syntax

```
call ssyevd (job,uplo,n,a, lda,w,work, lwork,iwork, liwork,info)
call dsyevd (job,uplo,n,a, lda,w,work, lwork,iwork, liwork,info)
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix A. In other words, it can compute the spectral factorization of A as: $A = Z\Lambda Z^T$.
Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i, and Z is the orthogonal matrix whose columns are the eigenvectors z_i. Thus,

$$Az_i = \lambda_i z_i \text{ for } i = 1, 2, \ldots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

job — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *job* = `'N'`, then only eigenvalues are computed.
If *job* = `'V'`, then eigenvalues and eigenvectors are computed.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, *a* stores the upper triangular part of A.
If *uplo* = `'L'`, *a* stores the lower triangular part of A.

n — `INTEGER`. The order of the matrix A ($n \geq 0$).

a — `REAL` for `ssyevd`
`DOUBLE PRECISION` for `dsyevd`
Array, `DIMENSION (lda, *)`.
`a(lda,*)` is an array containing either upper or lower triangular part of the symmetric matrix A, as specified by *uplo*.
The second dimension of *a* must be at least max(1, n).

lda	`INTEGER`. The first dimension of the array *a*. Must be at least max(1, *n*) .
work	`REAL` for `ssyevd` `DOUBLE PRECISION` for `dsyevd`. Workspace array, `DIMENSION` at least *lwork*.
lwork	`INTEGER`. The dimension of the array *work*. Constraints: if $n \le 1$, then *lwork* ≥ 1; if *job* = `'N'` and $n > 1$, then *lwork* ≥ 2*n*+1; if *job* = `'V'` and $n > 1$, then *lwork* ≥ $3n^2+(5+2k)*n+1$, where *k* is the smallest integer which satisfies $2^k \ge n$.
iwork	`INTEGER`. Workspace array, `DIMENSION` at least *liwork*.
liwork	`INTEGER`. The dimension of the array *iwork*. Constraints: if $n \le 1$, then *liwork* ≥ 1; if *job* = `'N'` and $n > 1$, then *liwork* ≥ 1; if *job* = `'V'` and $n > 1$, then *liwork* ≥ 5*n*+2.

Output Parameters

w	`REAL` for `ssyevd` `DOUBLE PRECISION` for `dsyevd` Array, `DIMENSION` at least max(1, *n*) . If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.
a	If *job* = `'V'`, then on exit this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors of *A*.
work(1)	On exit, if *lwork* > 0, then *work(1)* returns the required minimal size of *lwork*.
iwork(1)	On exit, if *liwork* > 0, then *iwork(1)* returns the required minimal size of *liwork*.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $\|E\|_2 = O(\varepsilon) \|T\|_2$, where ε is the machine precision.

The complex analogue of this routine is ?heevd.

?heevd

Computes all eigenvalues and (optionally) all eigenvectors of a complex Hermitian matrix using divide and conquer algorithm.

Syntax

```
call cheevd (job, uplo, n, a, lda, w, work, lwork, rwork, lrwork,
   iwork, liwork, info)
call zheevd (job, uplo, n, a, lda, w, work, lwork, rwork, lrwork,
   iwork, liwork, info)
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix A. In other words, it can compute the spectral factorization of A as: $A = Z\Lambda Z^H$. Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i, and Z is the (complex) unitary matrix whose columns are the eigenvectors z_i. Thus,

$$Az_i = \lambda_i z_i \text{ for } i = 1, 2, \ldots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

Input Parameters

job — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *job* =`'N'`, then only eigenvalues are computed.
If *job* =`'V'`, then eigenvalues and eigenvectors are computed.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, *a* stores the upper triangular part of *A*.
If *uplo* = `'L'`, *a* stores the lower triangular part of *A*.

n — `INTEGER`. The order of the matrix *A* ($n \geq 0$).

a — `COMPLEX` for `cheevd`
`DOUBLE COMPLEX` for `zheevd`
Array, `DIMENSION` (*lda*, *).
`a(`*lda*`,*)` is an array containing either upper or lower triangular part of the Hermitian matrix *A*, as specified by *uplo*.
The second dimension of *a* must be at least max(1, *n*).

lda — `INTEGER`. The first dimension of the array *a*.
Must be at least max(1, *n*) .

work — `COMPLEX` for `cheevd`
`DOUBLE COMPLEX` for `zheevd`.
Workspace array, `DIMENSION` at least *lwork*.

lwork — `INTEGER`. The dimension of the array *work*.
Constraints:
if $n \leq 1$, then $lwork \geq 1$;
if *job* =`'N'` and $n > 1$, then $lwork \geq n+1$;
if *job* =`'V'` and $n > 1$, then $lwork \geq n^2+2n$

rwork — `REAL` for `cheevd`
`DOUBLE PRECISION` for `zheevd`
Workspace array, `DIMENSION` at least *lrwork*.

lrwork — `INTEGER`. The dimension of the array *rwork*.
Constraints:
if $n \leq 1$, then $lrwork \geq 1$;
if *job* =`'N'` and $n > 1$, then $lrwork \geq n$;
if *job* =`'V'` and $n > 1$, then
$lrwork \geq 3n^2+(4+2k)*n+1$, where *k* is the smallest integer which satisfies $2^k \geq n$.

iwork — `INTEGER`.
Workspace array, `DIMENSION` at least *liwork*.

liwork — INTEGER. The dimension of the array *iwork*.
Constraints:
if $n \le 1$, then *liwork* ≥ 1;
if *job* = 'N' and $n > 1$, then *liwork* ≥ 1;
if *job* = 'V' and $n > 1$, then *liwork* $\ge 5n+2$.

Output Parameters

w — REAL for cheevd
DOUBLE PRECISION for zheevd
Array, DIMENSION at least max(1, *n*) .
If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order.
See also *info*.

a — If *job* = 'V', then on exit this array is overwritten by the unitary matrix *Z* which contains the eigenvectors of *A*.

work(1) — On exit, if *lwork* > 0, then the real part of *work(1)* returns the required minimal size of *lwork*.

rwork(1) — On exit, if *lrwork* > 0, then *rwork(1)* returns the required minimal size of *lrwork*.

iwork(1) — On exit, if *liwork* > 0, then *iwork(1)* returns the required minimal size of *liwork*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.
If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A + E$ such that $\|E\|_2 = O(\varepsilon)\|A\|_2$, where ε is the machine precision.

The real analogue of this routine is ?syevd.
See also ?hpevd for matrices held in packed storage, and ?hbevd for banded matrices.

?syevx

Computes selected eigenvalues and, optionally, eigenvectors of a symmetric matrix.

Syntax

```
call ssyevx (jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol,
             m, w, z, ldz, work, lwork, iwork, ifail, info)
call dsyevx (jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol,
             m, w, z, ldz, work, lwork, iwork, ifail, info)
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz
: `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then only eigenvalues are computed.
If *jobz* = `'V'`, then eigenvalues and eigenvectors are computed.

range
: `CHARACTER*1`. Must be `'A'`, `'V'`, or `'I'`.
If *range* = `'A'`, all eigenvalues will be found.
If *range* = `'V'`, all eigenvalues in the half-open interval (*vl*, *vu*] will be found.
If *range* = `'I'`, the eigenvalues with indices *il* through *iu* will be found.

uplo
: `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, *a* stores the upper triangular part of *A*.
If *uplo* = `'L'`, *a* stores the lower triangular part of *A*.

n
: `INTEGER`. The order of the matrix *A* ($n \geq 0$).

a, work
: `REAL` for `ssyevx`
`DOUBLE PRECISION` for `dsyevx`.
Arrays:
`a(lda,*)` is an array containing either upper or lower triangular part of the symmetric matrix *A*, as specified by *uplo*.
The second dimension of *a* must be at least max(1, *n*).

	`work(lwork)` is a workspace array.
lda	`INTEGER`. The first dimension of the array *a*. Must be at least max(1, *n*) .
vl, vu	`REAL` for `ssyevx` `DOUBLE PRECISION` for `dsyevx`. If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; $vl \leq vu$. Not referenced if *range* = 'A' or 'I'.
il, iu	`INTEGER`. If *range* = 'I', the indices of the smallest and largest eigenvalues to be returned. Constraints: $1 \leq il \leq iu \leq n$, if $n > 0$; $il = 1$ and $iu = 0$, if $n = 0$. Not referenced if *range* = 'A' or 'V'.
abstol	`REAL` for `ssyevx` `DOUBLE PRECISION` for `dsyevx`. The absolute error tolerance for the eigenvalues . See *Application notes* for more information.
ldz	`INTEGER`. The first dimension of the output array *z*; $ldz \geq 1$. If *jobz* = 'V', then $ldz \geq \max(1,n)$.
lwork	`INTEGER`. The dimension of the array *work*. Constraint: $lwork \geq \max(1, 8n)$. See *Application notes* for the suggested value of *lwork*.
iwork	`INTEGER`. Workspace array, `DIMENSION` at least max(1, 5*n*).

Output Parameters

a	On exit, the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.
m	`INTEGER`. The total number of eigenvalues found; $0 \leq m \leq n$. If *range* = 'A', $m = n$, and if *range* = 'I', $m = iu-il+1$.
w	`REAL` for `ssyevx` `DOUBLE PRECISION` for `dsyevx` Array, `DIMENSION` at least max(1, *n*) . The first *m* elements contain the selected eigenvalues of the matrix *A* in ascending order.

z
REAL for ssyevx
DOUBLE PRECISION for dsyevx.
Array *z*(*ldz*,*) contains eigenvectors.
The second dimension of *z* must be at least max(1, *m*).

If *jobz* ='V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the i-th column of *z* holding the eigenvector associated with *w*(i). If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
If *jobz* ='N', then *z* is not referenced.
Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* ='V', the exact value of *m* is not known in advance and an upper bound must be used.

work(1)
On exit, if *lwork* > 0, then *work(1)* returns the required minimal size of *lwork*.

ifail
INTEGER. Array, DIMENSION at least max(1, *n*).
If *jobz* ='V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, then *ifail* contains the indices of the eigenvectors that failed to converge.
If *jobz* ='V', then *ifail* is not referenced.

info
INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Application Notes

For optimum performance use *lwork* ≥ (*nb*+3)*n, where *nb* is the maximum of the blocksize for ?sytrd and ?ormtr returned by ilaenv.
If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to
abstol + ε * max(|a|,|b|) , where ε is the machine precision. If *abstol* is less than or equal to zero, then ε*|*T*| will be used in its place, where |*T*| is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*`slamch`('S'), not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*`slamch`('S').

?heevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

```
call cheevx (jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol,
             m, w, z, ldz, work, lwork, rwork, iwork, ifail, info)
call zheevx (jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol,
             m, w, z, ldz, work, lwork, rwork, iwork, ifail, info)
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then only eigenvalues are computed.
If *jobz* = `'V'`, then eigenvalues and eigenvectors are computed.

range — `CHARACTER*1`. Must be `'A'`, `'V'`, or `'I'`.
If *range* = `'A'`, all eigenvalues will be found.
If *range* = `'V'`, all eigenvalues in the half-open interval (*vl*, *vu*] will be found.
If *range* = `'I'`, the eigenvalues with indices *il* through *iu* will be found.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, *a* stores the upper triangular part of *A*.
If *uplo* = `'L'`, *a* stores the lower triangular part of *A*.

n — `INTEGER`. The order of the matrix *A* ($n \geq 0$).

a, work	`COMPLEX` for `cheevx` `DOUBLE COMPLEX` for `zheevx`. Arrays: `a(lda,*)` is an array containing either upper or lower triangular part of the Hermitian matrix *A*, as specified by *uplo*. The second dimension of `a` must be at least max(1, *n*). *work*(*lwork*) is a workspace array.
lda	`INTEGER`. The first dimension of the array `a`. Must be at least max(1, *n*) .
vl, vu	`REAL` for `cheevx` `DOUBLE PRECISION` for `zheevx`. If *range* = `'V'`, the lower and upper bounds of the interval to be searched for eigenvalues; $vl \le vu$. Not referenced if *range* = `'A'` or `'I'`.
il, iu	`INTEGER`. If *range* = `'I'`, the indices of the smallest and largest eigenvalues to be returned. Constraints: $1 \le il \le iu \le n$, if $n > 0$; $il = 1$ and $iu = 0$, if $n = 0$. Not referenced if *range* = `'A'` or `'V'`.
abstol	`REAL` for `cheevx` `DOUBLE PRECISION` for `zheevx`. The absolute error tolerance for the eigenvalues . See *Application notes* for more information.
ldz	`INTEGER`. The first dimension of the output array *z*; $ldz \ge 1$. If *jobz* = `'V'`, then $ldz \ge \max(1,n)$.
lwork	`INTEGER`. The dimension of the array *work*. Constraint: $lwork \ge \max(1, 2n-1)$. See *Application notes* for the suggested value of *lwork*.
rwork	`REAL` for `cheevx` `DOUBLE PRECISION` for `zheevx`. Workspace array, `DIMENSION` at least max(1, 7*n*).
iwork	`INTEGER`. Workspace array, `DIMENSION` at least max(1, 5*n*).

Output Parameters

a	On exit, the lower triangle (if *uplo* = `'L'`) or the upper triangle (if *uplo* = `'U'`) of *A*, including the diagonal, is overwritten.

m — INTEGER. The total number of eigenvalues found;
$0 \le m \le n$. If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu*-*il*+1.

w — REAL for cheevx
DOUBLE PRECISION for zheevx
Array, DIMENSION at least max(1, *n*).
The first *m* elements contain the selected eigenvalues of the matrix *A* in ascending order.

z — COMPLEX for cheevx
DOUBLE COMPLEX for zheevx.
Array *z*(*ldz*,*) contains eigenvectors.
The second dimension of *z* must be at least max(1, *m*).

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the i-th column of *z* holding the eigenvector associated with *w*(i). If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
If *jobz* = 'N', then *z* is not referenced.
Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

work(1) — On exit, if *lwork* > 0, then *work(1)* returns the required minimal size of *lwork*.

ifail — INTEGER. Array, DIMENSION at least max(1, *n*).
If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, then *ifail* contains the indices of the eigenvectors that failed to converge.
If *jobz* = 'V', then *ifail* is not referenced.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Application Notes

For optimum performance use `lwork` ≥ (*nb*+1)*`n`, where *nb* is the maximum of the blocksize for `?hetrd` and `?unmtr` returned by `ilaenv`.
If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to
`abstol` + ε * max(|a|,|b|) , where ε is the machine precision. If `abstol` is less than or equal to zero, then ε*|*T*| will be used in its place, where |*T*| is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form.
Eigenvalues will be computed most accurately when `abstol` is set to twice the underflow threshold 2*`slamch`('S'), not zero. If this routine returns with `info` > 0, indicating that some eigenvectors did not converge, try setting `abstol` to 2*`slamch`('S').

?syevr

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix using the Relatively Robust Representations.

Syntax

```
call ssyevr (jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol,
             m, w, z, ldz, isuppz, work, lwork, iwork, liwork, info)
call dsyevr (jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol,
             m, w, z, ldz, isuppz, work, lwork, iwork, liwork, info)
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *T*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Whenever possible, `?syevr` calls `sstegr/dstegr` to compute the eigenspectrum using Relatively Robust Representations. `?stegr` computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various “good" LDL^T representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the i-th unreduced block of *T*,

(a) Compute $T - \sigma_i = L_i D_i L_i^T$, such that $L_i D_i L_i^T$ is a relatively robust representation;

(b) Compute the eigenvalues, λ_j, of $L_i D_i L_i^T$ to high relative accuracy by the *dqds* algorithm;

(c) If there is a cluster of close eigenvalues, "choose" σ_i close to the cluster, and go to step (a);

(d) Given the approximate eigenvalue λ_j of $L_i D_i L_i^T$, compute the corresponding eigenvector by forming a rank-revealing twisted factorization.

The desired accuracy of the output can be specified by the input parameter *`abstol`*.

The routine `?syevr` calls `sstegr/dstegr` when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard. `?syevr` calls `sstebz/dstebz` and `sstein/dstein` on non-IEEE machines and when partial spectrum requests are made.

Input Parameters

jobz
`CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then only eigenvalues are computed.
If *jobz* = `'V'`, then eigenvalues and eigenvectors are computed.

range
`CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`.
If *range* = `'A'`, the routine computes all eigenvalues.
If *range* = `'V'`, the routine computes eigenvalues λ_i in the half-open interval: $vl < \lambda_i \le vu$.
If *range* = `'I'`, the routine computes eigenvalues with indices *il* to *iu*.

For *range* = `'V'` or `'I'` and $iu - il < n-1$, `sstebz/dstebz` and `sstein/dstein` are called.

n
`INTEGER`. The order of the matrix *A* ($n \ge 0$).

a, work
`REAL` for `ssyevr`
`DOUBLE PRECISION` for `dsyevr`.
Arrays:
a(*lda*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A*, as specified by *uplo*.
The second dimension of *a* must be at least max(1, *n*).

work(*lwork*) is a workspace array.

lda
`INTEGER`. The first dimension of the array *a*.
Must be at least max(1, *n*) .

vl, vu
`REAL` for `ssyevr`
`DOUBLE PRECISION` for `dsyevr`.
If *range* = `'V'`, the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: $vl < vu$.

If *range* = `'A'` or `'I'`, *vl* and *vu* are not referenced.

il, iu
`INTEGER`.
If *range* = `'I'`, the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: $1 \le il \le iu \le n$, if $n > 0$; *il*=1 and *iu*=0 if $n = 0$.

If *range* = `'A'` or `'V'`, *il* and *iu* are not referenced.

abstol REAL for ssyevr
DOUBLE PRECISION for dsyevr.
The absolute error tolerance to which each eigenvalue/eigenvector is required. If *jobz* = 'V', the eigenvalues and eigenvectors output have residual norms bounded by *abstol*, and the dot products between different eigenvectors are bounded by *abstol*. If *abstol* $< n\varepsilon\|T\|_1$, then $n\varepsilon\|T\|_1$ will be used in its place, where ε is the machine precision. The eigenvalues are computed to an accuracy of $\varepsilon\|T\|_1$ irrespective of *abstol*. If high relative accuracy is important, set *abstol* to ?lamch('S').

ldz INTEGER. The leading dimension of the output array *z*. Constraints:
ldz ≥ 1 if *jobz* ='N';
ldz $\geq \max(1, n)$ if *jobz* ='V'.

lwork INTEGER. The dimension of the array *work*.
Constraint: *lwork* $\geq \max(1, 26n)$. See *Application notes* for the suggested value of *lwork*.

iwork INTEGER.
Workspace array, DIMENSION (*liwork*).

liwork INTEGER. The dimension of the array *iwork*,
lwork $\geq \max(1, 10n)$.

Output Parameters

a On exit, the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

m INTEGER. The total number of eigenvalues found,
$0 \leq m \leq n$. If *range* ='A', $m = n$, and if *range* ='I',
$m = iu - il + 1$.

w, z REAL for ssyevr
DOUBLE PRECISION for dsyevr.
Arrays:
w(*), DIMENSION at least max(1, *n*), contains the selected eigenvalues in ascending order, stored in *w*(1) to *w*(*m*);

z(*ldz*, *), the second dimension of *z* must be at least max(1, *m*).
If *jobz* ='V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
If *jobz* ='N', then *z* is not referenced.

Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z* ; if *range* ='V', the exact value of *m* is not known in advance and an upper bound must be used.

isuppz INTEGER.
Array, DIMENSION at least 2*max(1, *m*).

The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th eigenvector is nonzero only in elements *isuppz*(2*i*-1) through *isuppz*(2*i*).
Implemented only for *range* ='A' or 'I' and
iu-*il* = *n*-1.

work(1) On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

iwork(1) On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, an internal error has occurred.

Application Notes

For optimum performance use *lwork* ≥ (*nb*+6)*n, where *nb* is the maximum of the blocksize for ?sytrd and ?ormtr returned by ilaenv.
If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

Normal execution of ?stegr may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

?heevr

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix using the Relatively Robust Representations.

Syntax

```
call cheevr ( jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol,
              m, w, z, ldz, isuppz, work, lwork, rwork, lrwork,
               iwork, liwork, info)
call zheevr ( jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol,
              m, w, z, ldz, isuppz, work, lwork, rwork, lrwork,
               iwork, liwork, info)
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *T*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Whenever possible, `?heevr` calls `cstegr/zstegr` to compute the eigenspectrum using Relatively Robust Representations. `?stegr` computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various “good" LDL^T representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the i-th unreduced block of *T*,

(a) Compute $T - \sigma_i = L_i D_i L_i^T$, such that $L_i D_i L_i^T$ is a relatively robust representation;

(b) Compute the eigenvalues, λ_j, of $L_i D_i L_i^T$ to high relative accuracy by the *dqds* algorithm;

(c) If there is a cluster of close eigenvalues, "choose" σ_i close to the cluster, and go to step (a);

(d) Given the approximate eigenvalue λ_j of $L_i D_i L_i^T$, compute the corresponding eigenvector by forming a rank-revealing twisted factorization.

The desired accuracy of the output can be specified by the input parameter `abstol`.

The routine ?heevr calls cstegr/zstegr when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard. ?heevr calls sstebz/dstebz and cstein/zstein on non-IEEE machines and when partial spectrum requests are made.

Input Parameters

jobz
: CHARACTER*1. Must be 'N' or 'V'.
If *job* ='N', then only eigenvalues are computed.
If *job* ='V', then eigenvalues and eigenvectors are computed.

range
: CHARACTER*1. Must be 'A' or 'V' or 'I'.
If *range* ='A', the routine computes all eigenvalues.
If *range* ='V', the routine computes eigenvalues λ_i in the half-open interval: $vl < \lambda_i \le vu$.
If *range* ='I', the routine computes eigenvalues with indices *il* to *iu*.

For *range* ='V'or 'I', sstebz/dstebz and cstein/zstein are called.

n
: INTEGER. The order of the matrix *A* ($n \ge 0$).

a, work
: COMPLEX for cheevr
DOUBLE COMPLEX for zheevr.
Arrays:
a(*lda*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A*, as specified by *uplo*.
The second dimension of a must be at least max(1, *n*).

work(*lwork*) is a workspace array.

lda
: INTEGER. The first dimension of the array a.
Must be at least max(1, *n*) .

vl, vu
: REAL for cheevr
DOUBLE PRECISION for zheevr.
If *range* ='V', the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: $vl < vu$.

If *range* ='A' or 'I', *vl* and *vu* are not referenced.

il, iu
: INTEGER.
If *range* ='I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: $1 \le il \le iu \le n$, if $n > 0$; *il*=1 and *iu*=0 if $n = 0$.

If *range* ='A' or 'V', *il* and *iu* are not referenced.

abstol — REAL for cheevr
DOUBLE PRECISION for zheevr.
The absolute error tolerance to which each eigenvalue/eigenvector is required. If *jobz* = 'V', the eigenvalues and eigenvectors output have residual norms bounded by *abstol*, and the dot products between different eigenvectors are bounded by *abstol*. If *abstol* $< n\varepsilon\|T\|_1$, then $n\varepsilon\|T\|_1$ will be used in its place, where ε is the machine precision. The eigenvalues are computed to an accuracy of $\varepsilon\|T\|_1$ irrespective of *abstol*. If high relative accuracy is important, set *abstol* to ?lamch('S').

ldz — INTEGER. The leading dimension of the output array *z*. Constraints:
ldz ≥ 1 if *jobz* ='N';
ldz $\geq \max(1, n)$ if *jobz* ='V'.

lwork — INTEGER. The dimension of the array *work*.
Constraint: *lwork* $\geq \max(1, 2n)$. See *Application notes* for the suggested value of *lwork*.

rwork — REAL for cheevr
DOUBLE PRECISION for zheevr.
Workspace array, DIMENSION (*lrwork*).

lrwork — INTEGER. The dimension of the array *rwork*;
lwork $\geq \max(1, 24n)$. .

iwork — INTEGER.
Workspace array, DIMENSION (*liwork*).

liwork — INTEGER. The dimension of the array *iwork*,
lwork $\geq \max(1, 10n)$.

Output Parameters

a — On exit, the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

m — INTEGER. The total number of eigenvalues found,
$0 \leq m \leq n$. If *range* ='A', $m = n$, and if *range* ='I',
$m = iu - il + 1$.

w — REAL for cheevr
DOUBLE PRECISION for zheevr.
Array, DIMENSION at least max(1, *n*), contains the selected eigenvalues in ascending order, stored in *w*(1) to *w*(*m*).

z — COMPLEX for cheevr
DOUBLE COMPLEX for zheevr.
Array *z*(*ldz*, *); the second dimension of *z* must be at least max(1, *m*).
If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
If *jobz* = 'N', then *z* is not referenced.
Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z* ; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

isuppz — INTEGER.
Array, DIMENSION at least 2*max(1, *m*).

The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th eigenvector is nonzero only in elements *isuppz*(2*i*-1) through *isuppz*(2*i*).

work(1) — On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

rwork(1) — On exit, if *info* = 0, then *rwork(1)* returns the required minimal size of *lrwork*.

iwork(1) — On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, an internal error has occurred.

Application Notes

For optimum performance use *lwork* ≥ (*nb*+1)***n*, where *nb* is the maximum of the blocksize for ?hetrd and ?unmtr returned by ilaenv.
If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

Normal execution of ?stegr may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

?spev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix in packed storage.

Syntax

```
call sspev (jobz, uplo, n, ap, w, z, ldz, work, info)
call dspev (jobz, uplo, n, ap, w, z, ldz, work, info)
```

Description

This routine computes all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A* in packed storage.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *job* ='N', then only eigenvalues are computed. If *job* ='V', then eigenvalues and eigenvectors are computed.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ap* stores the packed upper triangular part of *A*. If *uplo* = 'L', *ap* stores the packed lower triangular part of *A*.
n	INTEGER. The order of the matrix *A* ($n \ge 0$).
ap, *work*	REAL for sspev DOUBLE PRECISION for dspev Arrays: *ap(*)* contains the packed upper or lower triangle of symmetric matrix A, as specified by *uplo*. The dimension of *ap* must be at least max(1, $n*(n+1)/2$). *work(*)* is a workspace array, DIMENSION at least max(1, 3*n*).
ldz	INTEGER. The leading dimension of the output array *z*. Constraints: if *jobz* ='N', then $ldz \ge 1$; if *jobz* ='V', then $ldz \ge \max(1, n)$.

Output Parameters

w, *z*	REAL for sspev DOUBLE PRECISION for dspev Arrays: *w*(*), DIMENSION at least max(1, *n*) . If *info* = 0, *w* contains the eigenvalues of the matrix *A* in ascending order. z(*ldz*, *) . The second dimension of *z* must be at least max(1, *n*) . If *jobz* ='V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). If *jobz* ='N', then *z* is not referenced.
ap	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

?hpev

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian matrix in packed storage.

Syntax

```
call chpev (jobz, uplo, n, ap, w, z, ldz, work, rwork, info)
call zhpev (jobz, uplo, n, ap, w, z, ldz, work, rwork, info)
```

Description

This routine computes all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A* in packed storage.

Input Parameters

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *job* = `'N'`, then only eigenvalues are computed.
If *job* = `'V'`, then eigenvalues and eigenvectors are computed.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, *ap* stores the packed upper triangular part of *A*.
If *uplo* = `'L'`, *ap* stores the packed lower triangular part of *A*.

n — `INTEGER`. The order of the matrix *A* ($n \geq 0$).

ap, *work* — `COMPLEX` for `chpev`
`DOUBLE COMPLEX` for `zhpev`.
Arrays:
ap(*) contains the packed upper or lower triangle of Hermitian matrix A, as specified by *uplo*. The dimension of *ap* must be at least max(1, $n*(n+1)/2$).

work(*) is a workspace array, `DIMENSION` at least max(1, $2n-1$).

ldz — `INTEGER`. The leading dimension of the output array *z*.
Constraints:
if *jobz* = `'N'`, then $ldz \geq 1$;
if *jobz* = `'V'`, then $ldz \geq \max(1, n)$.

rwork — `REAL` for `chpev`
`DOUBLE PRECISION` for `zhpev`.
Workspace array, `DIMENSION` at least max(1, $3n-2$).

Output Parameters

w
`REAL` for `chpev`
`DOUBLE PRECISION` for `zhpev`.
Array, `DIMENSION` at least max(1, *n*).
If *info* = 0, *w* contains the eigenvalues of the matrix *A* in ascending order.

z
`COMPLEX` for `chpev`
`DOUBLE COMPLEX` for `zhpev`.
Array `z(`*ldz*, `*`). The second dimension of *z* must be at least max(1, *n*).
If *jobz* = `'V'`, then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
If *jobz* = `'N'`, then *z* is not referenced.

ap
On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A.

info
`INTEGER`.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

?spevd

Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a real symmetric matrix held in packed storage.

Syntax

```
call sspevd (job,uplo,n,ap,w,z,ldz,work,lwork,iwork,liwork,info)
call dspevd (job,uplo,n,ap,w,z,ldz,work,lwork,iwork,liwork,info)
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix A (held in packed storage). In other words, it can compute the spectral factorization of A as: $A = Z\Lambda Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i, and Z is the orthogonal matrix whose columns are the eigenvectors z_i. Thus,

$$Az_i = \lambda_i z_i \text{ for } i = 1, 2, \ldots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

Input Parameters

job	`CHARACTER*1`. Must be `'N'` or `'V'`. If *job* =`'N'`, then only eigenvalues are computed. If *job* =`'V'`, then eigenvalues and eigenvectors are computed.
uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. If *uplo* = `'U'`, *ap* stores the packed upper triangular part of A. If *uplo* = `'L'`, *ap* stores the packed lower triangular part of A.
n	`INTEGER`. The order of the matrix A ($n \geq 0$).
ap,*work*	`REAL` for `sspevd` `DOUBLE PRECISION` for `dspevd` Arrays: *ap(*)* contains the packed upper or lower triangle of symmetric matrix A, as specified by *uplo*. The dimension of *ap* must be at least max(1, $n*(n+1)/2$) *work(*)* is a workspace array, `DIMENSION` at least *lwork*.
ldz	`INTEGER`. The leading dimension of the output array *z*. Constraints: if *job* =`'N'`, then *ldz* ≥ 1; if *job* =`'V'`, then *ldz* $\geq$ max(1, n) .
lwork	`INTEGER`. The dimension of the array *work*. Constraints: if $n \leq 1$, then *lwork* ≥ 1; if *job* =`'N'` and $n > 1$, then *lwork* $\geq 2n$; if *job* =`'V'` and $n > 1$, then *lwork* $\geq 2n^2+(5+2k)*n+1$, where k is the smallest integer which satisfies $2^k \geq n$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by `xerbla`.

iwork — `INTEGER`.
Workspace array, `DIMENSION` at least *liwork*.

liwork — `INTEGER`. The dimension of the array *iwork*.
Constraints:
if $n \le 1$, then *liwork* ≥ 1;
if *job* = 'N' and $n > 1$, then *liwork* ≥ 1;
if *job* = 'V' and $n > 1$, then *liwork* ≥ $5n+3$.
If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by `xerbla`.

Output Parameters

w,z — `REAL` for `sspevd`
`DOUBLE PRECISION` for `dspevd`
Arrays:
w(*), `DIMENSION` at least max(1, *n*) .
If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.
z(ldz,)* . The second dimension of *z* must be:
at least 1 if *job* = 'N';
at least max(1, *n*) if *job* = 'V'.
If *job* = 'V', then this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors of *A*. If *job* = 'N', then *z* is not referenced.

ap — On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A.

work(1) — On exit, if *info* = 0, then *work(1)* returns the optimal *lwork*.

iwork(1) — On exit, if *info* = 0, then *iwork(1)* returns the optimal *liwork*.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.
If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $\|E\|_2 = O(\varepsilon)\ \|T\|_2$, where ε is the machine precision.

The complex analogue of this routine is `?hpevd`.

See also ?syevd for matrices held in full storage, and ?sbevd for banded matrices.

?hpevd

Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a complex Hermitian matrix held in packed storage.

Syntax

```
call chpevd (job, uplo, n, ap, w, z, ldz, work, lwork, rwork,
   lrwork, iwork, liwork, info)
call zhpevd (job, uplo, n, ap, w, z, ldz, work, lwork, rwork,
   lrwork, iwork, liwork, info)
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix A (held in packed storage). In other words, it can compute the spectral factorization of A as: $A = Z\Lambda Z^H$.
Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i, and Z is the (complex) unitary matrix whose columns are the eigenvectors z_i. Thus,

$$Az_i = \lambda_i z_i \text{ for } i = 1, 2, \ldots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

job — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *job* = `'N'`, then only eigenvalues are computed.
If *job* = `'V'`, then eigenvalues and eigenvectors are computed.

uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ap* stores the packed upper triangular part of *A*. If *uplo* = 'L', *ap* stores the packed lower triangular part of *A*.
n	INTEGER. The order of the matrix *A* ($n \ge 0$).
ap,work	COMPLEX for chpevd DOUBLE COMPLEX for zhpevd Arrays: *ap(*)* contains the packed upper or lower triangle of Hermitian matrix A, as specified by *uplo*. The dimension of *ap* must be at least max(1, $n*(n+1)/2$) *work(*)* is a workspace array, DIMENSION at least *lwork*.
ldz	INTEGER. The leading dimension of the output array *z*. Constraints: if *job* ='N', then $ldz \ge 1$; if *job* ='V', then $ldz \ge$ max(1, *n*) .
lwork	INTEGER. The dimension of the array *work*. Constraints: if $n \le 1$, then $lwork \ge 1$; if *job* ='N' and $n > 1$, then $lwork \ge n$; if *job* ='V' and $n > 1$, then $lwork \ge 2n$
rwork	REAL for chpevd DOUBLE PRECISION for zhpevd Workspace array, DIMENSION at least *lrwork*.
lrwork	INTEGER. The dimension of the array *rwork*. Constraints: if $n \le 1$, then $lrwork \ge 1$; if *job* ='N' and $n > 1$, then $lrwork \ge n$; if *job* ='V' and $n > 1$, then $lrwork \ge 3n^2+(4+2k)*n+1$, where *k* is the smallest integer which satisfies $2^k \ge n$.
iwork	INTEGER. Workspace array, DIMENSION at least *liwork*.
liwork	INTEGER. The dimension of the array *iwork*. Constraints: if $n \le 1$, then $liwork \ge 1$; if *job* ='N' and $n > 1$, then $liwork \ge 1$; if *job* ='V' and $n > 1$, then $liwork \ge 5n+2$.

Output Parameters

w	`REAL` for `chpevd` `DOUBLE PRECISION` for `zhpevd` Array, `DIMENSION` at least max(1, *n*) . If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.
z	`COMPLEX` for `chpevd` `DOUBLE COMPLEX` for `zhpevd` Array, `DIMENSION` *(ldz, *)* . The second dimension of *z* must be: at least 1 if *job* = `'N'`; at least max(1, *n*) if *job* = `'V'`. If *job* = `'V'`, then this array is overwritten by the unitary matrix *Z* which contains the eigenvectors of *A*. If *job* = `'N'`, then *z* is not referenced.
ap	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A.
work(1)	On exit, if *lwork* > 0, then the real part of *work(1)* returns the required minimal size of *lwork*.
rwork(1)	On exit, if *lrwork* > 0, then *rwork(1)* returns the required minimal size of *lrwork*.
iwork(1)	On exit, if *liwork* > 0, then *iwork(1)* returns the required minimal size of *liwork*.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero. If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $\|E\|_2 = O(\varepsilon)\|T\|_2$, where ε is the machine precision.

The real analogue of this routine is `?spevd`.

See also ?heevd for matrices held in full storage, and ?hbevd for banded matrices.

?spevx

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix in packed storage.

Syntax

```
call sspevx (jobz, range, uplo, n, ap, vl, vu, il, iu, abstol,
             m, w, z, ldz, work, iwork, ifail, info)
call dspevx (jobz, range, uplo, n, ap, vl, vu, il, iu, abstol,
             m, w, z, ldz, work, iwork, ifail, info)
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A* in packed storage. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *job* = `'N'`, then only eigenvalues are computed.
If *job* = `'V'`, then eigenvalues and eigenvectors are computed.

range — `CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`.
If *range* = `'A'`, the routine computes all eigenvalues.
If *range* = `'V'`, the routine computes eigenvalues λ_i in the half-open interval: $vl < \lambda_i \le vu$.
If *range* = `'I'`, the routine computes eigenvalues with indices *il* to *iu*.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, *ap* stores the packed upper triangular part of *A*.
If *uplo* = `'L'`, *ap* stores the packed lower triangular part of *A*.

n — `INTEGER`. The order of the matrix *A* ($n \ge 0$).

ap, work — `REAL` for `sspevx`
`DOUBLE PRECISION` for `dspevx`
Arrays:
ap(*) contains the packed upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*. The dimension of *ap* must be at least max(1, $n*(n+1)/2$).

	`work(*)` is a workspace array, `DIMENSION` at least max(1, 8*n*).
vl, vu	`REAL` for `sspevx` `DOUBLE PRECISION` for `dspevx` If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: *vl* < *vu*. If *range* = 'A' or 'I', *vl* and *vu* are not referenced.
il, iu	`INTEGER`. If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \le il \le iu \le n$, if $n > 0$; *il*=1 and *iu*=0 if $n = 0$. If *range* = 'A' or 'V', *il* and *iu* are not referenced.
abstol	`REAL` for `sspevx` `DOUBLE PRECISION` for `dspevx` The absolute error tolerance to which each eigenvalue is required. See *Application notes* for details on error tolerance.
ldz	`INTEGER`. The leading dimension of the output array *z*. Constraints: if *jobz* = 'N', then $ldz \ge 1$; if *jobz* = 'V', then $ldz \ge \max(1, n)$.
iwork	`INTEGER`. Workspace array, `DIMENSION` at least max(1, 5*n*).

Output Parameters

ap	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of *A*.
m	`INTEGER`. The total number of eigenvalues found, $0 \le m \le n$. If *range* = 'A', $m = n$, and if *range* = 'I', $m = iu - il + 1$.
w, *z*	`REAL` for `sspevx` `DOUBLE PRECISION` for `dspevx` Arrays: `w(*)`, `DIMENSION` at least max(1, *n*). If *info* = 0, contains the selected eigenvalues of the matrix *A* in ascending order.

`z(ldz,*)`. The second dimension of *z* must be at least max(1, *m*).
If *jobz* ='V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
If *jobz* ='N', then *z* is not referenced.
Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* ='V', the exact value of *m* is not known in advance and an upper bound must be used.

ifail — INTEGER. Array, DIMENSION at least max(1, *n*).
If *jobz* ='V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices the eigenvectors that failed to converge.
If *jobz* ='N', then *ifail* is not referenced.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.
If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to
abstol + ε * max(|a|,|b|), where ε is the machine precision. If *abstol* is less than or equal to zero, then $\varepsilon * \|T\|_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form.
Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?hpevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix in packed storage.

Syntax

```
call chpevx (jobz, range, uplo, n, ap, vl, vu, il, iu, abstol,
             m, w, z, ldz, work, rwork, iwork, ifail, info)
call zhpevx (jobz, range, uplo, n, ap, vl, vu, il, iu, abstol,
             m, w, z, ldz, work, rwork, iwork, ifail, info)
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A* in packed storage. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *job* = `'N'`, then only eigenvalues are computed.
If *job* = `'V'`, then eigenvalues and eigenvectors are computed.

range — `CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`.
If *range* = `'A'`, the routine computes all eigenvalues.
If *range* = `'V'`, the routine computes eigenvalues λ_i in the half-open interval: $vl < \lambda_i \le vu$.
If *range* = `'I'`, the routine computes eigenvalues with indices *il* to *iu*.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, *ap* stores the packed upper triangular part of *A*.
If *uplo* = `'L'`, *ap* stores the packed lower triangular part of *A*.

n — `INTEGER`. The order of the matrix *A* ($n \ge 0$).

ap, work — `COMPLEX` for `chpevx`
`DOUBLE COMPLEX` for `zhpevx`
Arrays:
ap(*) contains the packed upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*. The dimension of *ap* must be at least max(1, $n*(n+1)/2$).

work(*) is a workspace array, `DIMENSION` at least max(1, 2*n*).

vl, vu — REAL for chpevx
DOUBLE PRECISION for zhpevx
If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: *vl* < *vu*.
If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, iu — INTEGER.
If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: $1 \le il \le iu \le n$, if $n > 0$; *il*=1 and *iu*=0 if $n = 0$.
If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol — REAL for chpevx
DOUBLE PRECISION for zhpevx
The absolute error tolerance to which each eigenvalue is required. See *Application notes* for details on error tolerance.

ldz — INTEGER. The leading dimension of the output array *z*.
Constraints:
if *jobz* = 'N', then $ldz \ge 1$;
if *jobz* = 'V', then $ldz \ge \max(1, n)$.

rwork — REAL for chpevx
DOUBLE PRECISION for zhpevx
Workspace array, DIMENSION at least max(1, 7*n*).

iwork — INTEGER.
Workspace array, DIMENSION at least max(1, 5*n*).

Output Parameters

ap — On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of *A*.

m — INTEGER. The total number of eigenvalues found, $0 \le m \le n$. If *range* = 'A', $m = n$, and if *range* = 'I', $m = iu - il + 1$.

w — REAL for chpevx
DOUBLE PRECISION for zhpevx
Array, DIMENSION at least max(1, *n*). If *info* = 0, contains the selected eigenvalues of the matrix *A* in ascending order.

z — COMPLEX for chpevx
DOUBLE COMPLEX for zhpevx
Array z(*ldz*,*) . The second dimension of *z* must be at least max(1, *m*) .
If *jobz* ='V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
If *jobz* ='N', then *z* is not referenced.
Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z* ; if *range* ='V', the exact value of *m* is not known in advance and an upper bound must be used.

ifail — INTEGER. Array, DIMENSION at least max(1, *n*).
If *jobz* ='V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices the eigenvectors that failed to converge.
If *jobz* ='N', then *ifail* is not referenced.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to
abstol + ε * max(|a|,|b|) , where ε is the machine precision. If *abstol* is less than or equal to zero, then $\varepsilon * \|T\|_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form.
Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?sbev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric band matrix.

Syntax

```
call ssbev (jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, info)
call dsbev (jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, info)
```

Description

This routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric band matrix *A*.

Input Parameters

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* =`'N'`, then only eigenvalues are computed.
If *jobz* =`'V'`, then eigenvalues and eigenvectors are computed.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, *ab* stores the upper triangular part of *A*.
If *uplo* = `'L'`, *ab* stores the lower triangular part of *A*.

n — `INTEGER`. The order of the matrix *A* ($n \geq 0$).

kd — `INTEGER`. The number of super- or sub-diagonals in *A* ($kd \geq 0$).

ab, work — `REAL` for `ssbev`
`DOUBLE PRECISION` for `dsbev`.
Arrays:
ab (*ldab*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format.
The second dimension of *ab* must be at least max(1, *n*).

work (*) is a workspace array.
The dimension of *work* must be at least max(1, 3*n*-2).

ldab — `INTEGER`. The leading dimension of *ab*; must be at least *kd* +1.

ldz INTEGER. The leading dimension of the output array *z*.
Constraints:
if *jobz* ='N', then $ldz \geq 1$;
if *jobz* ='V', then $ldz \geq \max(1, n)$.

Output Parameters

w,z REAL for ssbev
DOUBLE PRECISION for dsbev
Arrays:
w(*), DIMENSION at least $\max(1, n)$.
If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order.

z(ldz,)*. The second dimension of *z* must be at least max(1, *n*).
If *jobz* ='V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
If *jobz* ='N', then *z* is not referenced.

ab On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. If *uplo* = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix *T* are returned in rows *kd* and *kd*+1 of *ab*, and if *uplo* = 'L', the diagonal and first subdiagonal of *T* are returned in the first two rows of *ab*.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, then the algorithm failed to converge;
i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

?hbev

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian band matrix.

Syntax

```
call chbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, rwork,info)
call zhbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, rwork,info)
```

Description

This routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix *A*.

Input Parameters

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* =`'N'`, then only eigenvalues are computed.
If *jobz* =`'V'`, then eigenvalues and eigenvectors are computed.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, *ab* stores the upper triangular part of *A*.
If *uplo* = `'L'`, *ab* stores the lower triangular part of *A*.

n — `INTEGER`. The order of the matrix *A* ($n \geq 0$).

kd — `INTEGER`. The number of super- or sub-diagonals in *A* ($kd \geq 0$).

ab, work — `COMPLEX` for `chbev`
`DOUBLE COMPLEX` for `zhbev`.
Arrays:
ab (*ldab*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format.
The second dimension of *ab* must be at least max(1, *n*).

work (*) is a workspace array.
The dimension of *work* must be at least max(1, *n*).

ldab — `INTEGER`. The leading dimension of *ab*; must be at least *kd* +1.

ldz — `INTEGER`. The leading dimension of the output array *z*.
Constraints:
if *jobz* = `'N'`, then *ldz* ≥ 1;
if *jobz* = `'V'`, then *ldz* ≥ max(1, *n*) .

rwork — `REAL` for `chbev`
`DOUBLE PRECISION` for `zhbev`
Workspace array, `DIMENSION` at least max(1, 3*n*-2).

Output Parameters

w — `REAL` for `chbev`
`DOUBLE PRECISION` for `zhbev`
Array, `DIMENSION` at least max(1, *n*). If *info* = 0, contains the eigenvalues in ascending order.

z — `COMPLEX` for `chbev`
`DOUBLE COMPLEX` for `zhbev`.
Array *z(ldz,*)* . The second dimension of *z* must be at least max(1, *n*). If *jobz* = `'V'`, then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). If *jobz* = `'N'`, then *z* is not referenced.

ab — On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. If *uplo* = `'U'`, the first superdiagonal and the diagonal of the tridiagonal matrix *T* are returned in rows *kd* and *kd*+1 of *ab*, and if *uplo* = `'L'`, the diagonal and first subdiagonal of *T* are returned in the first two rows of *ab*.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.
If *info* = *i*, then the algorithm failed to converge;
i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

?sbevd

Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric band matrix using divide and conquer algorithm.

Syntax

```
call ssbevd (job, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork,
  iwork, liwork, info)
call dsbevd (job, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork,
  iwork, liwork, info)
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric band matrix A . In other words, it can compute the spectral factorization of A as:

$$A = Z\Lambda Z^T$$

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i, and Z is the orthogonal matrix whose columns are the eigenvectors z_i.
Thus,

$$Az_i = \lambda_i z_i \text{ for } i = 1, 2, \ldots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

job — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *job* = `'N'`, then only eigenvalues are computed.
If *job* = `'V'`, then eigenvalues and eigenvectors are computed.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, *ab* stores the upper triangular part of A.
If *uplo* = `'L'`, *ab* stores the lower triangular part of A.

n — `INTEGER`. The order of the matrix A ($n \ge 0$).

kd — `INTEGER`. The number of super- or sub-diagonals in A ($kd \ge 0$).

ab, work	`REAL` for `ssbevd` `DOUBLE PRECISION` for `dsbevd`. Arrays: *ab* (*ldab*, *) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format. The second dimension of *ab* must be at least max(1, *n*). *work* (*) is a workspace array. The dimension of *work* must be at least *lwork*.
ldab	`INTEGER`. The leading dimension of *ab*; must be at least *kd*+1.
ldz	`INTEGER`. The leading dimension of the output array *z*. Constraints: if *job* ='N', then $ldz \geq 1$; if *job* ='V', then $ldz \geq \max(1, n)$.
lwork	`INTEGER`. The dimension of the array *work*. Constraints: if $n \leq 1$, then $lwork \geq 1$; if *job* ='N' and $n > 1$, then $lwork \geq 2n$; if *job* ='V' and $n > 1$, then $lwork \geq 3n^2+(4+2k)*n+1$, where *k* is the smallest integer which satisfies $2^k \geq n$.
iwork	`INTEGER`. Workspace array, `DIMENSION` at least *liwork*.
liwork	`INTEGER`. The dimension of the array *iwork*. Constraints: if $n \leq 1$, then $liwork \geq 1$; if *job* ='N' and $n > 1$, then $liwork \geq 1$; if *job* ='V' and $n > 1$, then $liwork \geq 5n+2$.

Output Parameters

w, z	`REAL` for `ssbevd` `DOUBLE PRECISION` for `dsbevd` Arrays: *w*(*), `DIMENSION` at least max(1, *n*) . If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. See also *info*. *z*(*ldz*, *) . The second dimension of *z* must be: at least 1 if *job* ='N';

	at least max(1, *n*) if *job* = 'V'. If *job* = 'V', then this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors of *A*. The *i*th column of *Z* contains the eigenvector which corresponds to the eigenvalue *w(i)*. If *job* = 'N', then *z* is not referenced.
ab	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.
work(1)	On exit, if *lwork* > 0, then *work(1)* returns the required minimal size of *lwork*.
iwork(1)	On exit, if *liwork* > 0, then *iwork(1)* returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero. If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $\|E\|_2 = O(\varepsilon)\|T\|_2$, where ε is the machine precision.

The complex analogue of this routine is ?hbevd.

See also ?syevd for matrices held in full storage, and ?spevd for matrices held in packed storage.

?hbevd

Computes all eigenvalues and (optionally) all eigenvectors of a complex Hermitian band matrix using divide and conquer algorithm.

Syntax

```
call chbevd (job, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork,
  rwork, lrwork, iwork, liwork, info)
call zhbevd (job, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork,
  rwork, lrwork, iwork, liwork, info)
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian band matrix A . In other words, it can compute the spectral factorization of A as: $A = Z\Lambda Z^H$.
Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i, and Z is the (complex) unitary matrix whose columns are the eigenvectors z_i. Thus,

$$Az_i = \lambda_i z_i \text{ for } i = 1, 2, \ldots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

job — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *job* = `'N'`, then only eigenvalues are computed.
If *job* = `'V'`, then eigenvalues and eigenvectors are computed.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, *ab* stores the upper triangular part of A.
If *uplo* = `'L'`, *ab* stores the lower triangular part of A.

n — `INTEGER`. The order of the matrix A ($n \geq 0$).

kd — `INTEGER`. The number of super- or sub-diagonals in A ($kd \geq 0$).

ab, work — `COMPLEX` for `chbevd`
`DOUBLE COMPLEX` for `zhbevd`.
Arrays:
ab (*ldab*,*) is an array containing either upper or lower triangular part of the Hermitian matrix A (as specified by *uplo*) in band storage format.
The second dimension of *ab* must be at least max(1, *n*).

work (*) is a workspace array.
The dimension of *work* must be at least *lwork*.

ldab — `INTEGER`. The leading dimension of *ab*; must be at least *kd*+1.

ldz — `INTEGER`. The leading dimension of the output array *z*.
Constraints:
if *job* = `'N'`, then $ldz \geq 1$;
if *job* = `'V'`, then $ldz \geq \max(1, n)$.

lwork — INTEGER. The dimension of the array *work*.
Constraints:
if $n \le 1$, then $lwork \ge 1$;
if *job* = 'N' and $n > 1$, then $lwork \ge n$;
if *job* = 'V' and $n > 1$, then $lwork \ge 2n^2$

rwork — REAL for chbevd
DOUBLE PRECISION for zhbevd
Workspace array, DIMENSION at least *lrwork*.

lrwork — INTEGER. The dimension of the array *rwork*.
Constraints:
if $n \le 1$, then $lrwork \ge 1$;
if *job* = 'N' and $n > 1$, then $lrwork \ge n$;
if *job* = 'V' and $n > 1$, then
$lrwork \ge 3n^2+(4+2k)*n+1$, where k is the smallest integer which satisfies $2^k \ge n$.

iwork — INTEGER.
Workspace array, DIMENSION at least *liwork*.

liwork — INTEGER. The dimension of the array *iwork*.
Constraints:
if *job* = 'N' or $n \le 1$, then $liwork \ge 1$;
if *job* = 'V' and $n > 1$, then $liwork \ge 5n+2$.

Output Parameters

w — REAL for chbevd
DOUBLE PRECISION for zhbevd
Array, DIMENSION at least max(1, *n*) .
If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.

z — COMPLEX for chbevd
DOUBLE COMPLEX for zhbevd
Array, DIMENSION *(ldz, *)* . The second dimension of *z* must be:
at least 1 if *job* = 'N';
at least max(1, *n*) if *job* = 'V'.
If *job* = 'V', then this array is overwritten by the unitary matrix *Z* which contains the eigenvectors of *A*. The *i*th column of *Z* contains the eigenvector which corresponds to the eigenvalue *w(i)*.
If *job* = 'N', then *z* is not referenced.

ab	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.
work(1)	On exit, if *lwork* > 0, then the real part of *work(1)* returns the required minimal size of *lwork*.
rwork(1)	On exit, if *lrwork* > 0, then *rwork(1)* returns the required minimal size of *lrwork*.
iwork(1)	On exit, if *liwork* > 0, then *iwork(1)* returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero. If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $\|E\|_2 = O(\varepsilon)\|T\|_2$, where ε is the machine precision.

The real analogue of this routine is ?sbevd.

See also ?heevd for matrices held in full storage, and ?hpevd for matrices held in packed storage.

?sbevx

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix.

Syntax

```
call ssbevx ( jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il,
              iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
call dsbevx ( jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il,
              iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *job* = `'N'`, then only eigenvalues are computed.
If *job* = `'V'`, then eigenvalues and eigenvectors are computed.

range — `CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`.
If *range* = `'A'`, the routine computes all eigenvalues.
If *range* = `'V'`, the routine computes eigenvalues λ_i in the half-open interval: $vl < \lambda_i \le vu$.
If *range* = `'I'`, the routine computes eigenvalues with indices *il* to *iu*.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, *ab* stores the upper triangular part of *A*.
If *uplo* = `'L'`, *ab* stores the lower triangular part of *A*.

n — `INTEGER`. The order of the matrix *A* ($n \ge 0$).

kd — `INTEGER`. The number of super- or sub-diagonals in *A* ($kd \ge 0$).

ab, work — `REAL` for `ssbevx`
`DOUBLE PRECISION` for `dsbevx`.
Arrays:

	ab (*ldab*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format. The second dimension of *ab* must be at least max(1, *n*). *work* (*) is a workspace array. The dimension of *work* must be at least max(1, 7*n*).
ldab	INTEGER. The leading dimension of *ab*; must be at least *kd* +1.
vl, vu	REAL for ssbevx DOUBLE PRECISION for dsbevx. If *range* ='V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: *vl*< *vu*. If *range* ='A' or 'I', *vl* and *vu* are not referenced.
il, iu	INTEGER. If *range* ='I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: 1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0. If *range* ='A' or 'V', *il* and *iu* are not referenced.
abstol	REAL for chpevx DOUBLE PRECISION for zhpevx The absolute error tolerance to which each eigenvalue is required. See *Application notes* for details on error tolerance.
ldq, ldz	INTEGER. The leading dimensions of the output arrays *q* and *z*, respectively. Constraints: *ldq* ≥ 1, *ldz* ≥ 1; If *jobz* ='V', then *ldq* ≥ max(1, *n*) and *ldz* ≥ max(1, *n*).
iwork	INTEGER. Workspace array, DIMENSION at least max(1, 5*n*).

Output Parameters

m	INTEGER. The total number of eigenvalues found, 0 ≤ *m* ≤ *n*. If *range* ='A', *m* = *n*, and if *range* ='I', *m* = *iu*-*il*+1.
w,z	REAL for ssbevx DOUBLE PRECISION for dsbevx Arrays:

`w(*)`, `DIMENSION` at least max(1, *n*) .
The first *m* elements of *w* contain the selected eigenvalues of the matrix *A* in ascending order.

`z(ldz,*)` . The second dimension of *z* must be at least max(1, *m*).
If *jobz* ='V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
If *jobz* ='N', then *z* is not referenced.
Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z* ; if *range* ='V', the exact value of *m* is not known in advance and an upper bound must be used.

ab — On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. If *uplo* = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix *T* are returned in rows *kd* and *kd*+1 of *ab*, and if *uplo* = 'L', the diagonal and first subdiagonal of *T* are returned in the first two rows of *ab*.

ifail — `INTEGER`.
Array, `DIMENSION` at least max(1, *n*).
If *jobz* ='V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices the eigenvectors that failed to converge.
If *jobz* ='N', then *ifail* is not referenced.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to
abstol + ε * max(|a|,|b|) , where ε is the machine precision. If *abstol* is less than or equal to zero, then $\varepsilon * \|T\|_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?hbevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian band matrix.

Syntax

```
call chbevx ( jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il,
        iu, abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)
call zhbevx ( jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il,
        iu, abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz	`CHARACTER*1`. Must be `'N'` or `'V'`. If *job* = `'N'`, then only eigenvalues are computed. If *job* = `'V'`, then eigenvalues and eigenvectors are computed.
range	`CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`. If *range* = `'A'`, the routine computes all eigenvalues. If *range* = `'V'`, the routine computes eigenvalues λ_i in the half-open interval: $vl < \lambda_i \le vu$. If *range* = `'I'`, the routine computes eigenvalues with indices *il* to *iu*.
uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. If *uplo* = `'U'`, *ab* stores the upper triangular part of *A*. If *uplo* = `'L'`, *ab* stores the lower triangular part of *A*.
n	`INTEGER`. The order of the matrix *A* ($n \ge 0$).
kd	`INTEGER`. The number of super- or sub-diagonals in *A* ($kd \ge 0$).
ab, work	`COMPLEX` for `chbevx` `DOUBLE COMPLEX` for `zhbevx`. Arrays:

	`ab (ldab,*)` is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by `uplo`) in band storage format. The second dimension of `ab` must be at least max(1, `n`). `work (*)` is a workspace array. The dimension of `work` must be at least max(1, `n`).
`ldab`	`INTEGER`. The leading dimension of `ab`; must be at least `kd` +1.
`vl, vu`	`REAL` for `chbevx` `DOUBLE PRECISION` for `zhbevx`. If `range` ='V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: `vl`< `vu`. If `range` ='A' or 'I', `vl` and `vu` are not referenced.
`il, iu`	`INTEGER`. If `range` ='I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: 1 ≤ `il` ≤ `iu` ≤ `n`, if `n` > 0; `il`=1 and `iu`=0 if `n` = 0. If `range` ='A' or 'V', `il` and `iu` are not referenced.
`abstol`	`REAL` for `chbevx` `DOUBLE PRECISION` for `zhbevx`. The absolute error tolerance to which each eigenvalue is required. See *Application notes* for details on error tolerance.
`ldq, ldz`	`INTEGER`. The leading dimensions of the output arrays `q` and `z`, respectively. Constraints: `ldq` ≥ 1, `ldz` ≥ 1; If `jobz` ='V', then `ldq` ≥ max(1, `n`) and `ldz` ≥ max(1, `n`).
`rwork`	`REAL` for `chbevx` `DOUBLE PRECISION` for `zhbevx` Workspace array, `DIMENSION` at least max(1, 7`n`).
`iwork`	`INTEGER`. Workspace array, `DIMENSION` at least max(1, 5`n`).

Output Parameters

`m`	`INTEGER`. The total number of eigenvalues found, 0 ≤ `m` ≤ `n`. If `range` ='A', `m` = `n`, and if `range` ='I', `m` = `iu`-`il`+1.

w
: REAL for chbevx
DOUBLE PRECISION for zhbevx
Array, DIMENSION at least max(1, *n*) .
The first *m* elements contain the selected eigenvalues of the matrix *A* in ascending order.

z
: COMPLEX for chbevx
DOUBLE COMPLEX for zhbevx.
Array z*(ldz,*)* . The second dimension of *z* must be at least max(1, *m*).
If *jobz* ='V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
If *jobz* ='N', then *z* is not referenced.
Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z* ; if *range* ='V', the exact value of *m* is not known in advance and an upper bound must be used.

ab
: On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. If *uplo* = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix *T* are returned in rows *kd* and *kd*+1 of *ab*, and if *uplo* = 'L', the diagonal and first subdiagonal of *T* are returned in the first two rows of *ab*.

ifail
: INTEGER.
Array, DIMENSION at least max(1, *n*).
If *jobz* ='V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.
If *jobz* ='N', then *ifail* is not referenced.

info
: INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to

abstol + ε * max(|a|,|b|) , where ε is the machine precision. If *abstol* is less than or equal to zero, then $\varepsilon * \|T\|_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?stev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix.

Syntax

```
call sstev (jobz, n, d, e, z, ldz, work, info)
call dstev (jobz, n, d, e, z, ldz, work, info)
```

Description

This routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix *A*.

Input Parameters

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then only eigenvalues are computed.
If *jobz* = `'V'`, then eigenvalues and eigenvectors are computed.

n — `INTEGER`. The order of the matrix *A* ($n \geq 0$).

d, e, work — `REAL` for `sstev`
`DOUBLE PRECISION` for `dstev`.
Arrays:
d(*) contains the *n* diagonal elements of the tridiagonal matrix *A*.
The dimension of *d* must be at least max(1, *n*).

e(*) contains the *n*-1 subdiagonal elements of the tridiagonal matrix *A*.
The dimension of *e* must be at least max(1, *n*). The *n*th element of this array is used as workspace.

work(*) is a workspace array.
The dimension of *work* must be at least max(1, 2*n*-2).
If *jobz* = `'N'`, *work* is not referenced.

ldz — `INTEGER`. The leading dimension of the output array *z*; $ldz \geq 1$. If *jobz* = `'V'` then $ldz \geq \max(1, n)$.

Output Parameters

d — On exit, if *info* = 0, contains the eigenvalues of the matrix *A* in ascending order.

z — REAL for sstev
DOUBLE PRECISION for dstev
Array, DIMENSION (*ldz*, *).
The second dimension of *z* must be at least max(1, *n*).
If *jobz* ='V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with the eigenvalue returned in *d*(*i*).
If *job* ='N', then *z* is not referenced.

e — On exit, this array is overwritten with intermediate results.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, then the algorithm failed to converge;
i elements of *e* did not converge to zero.

?stevd

Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric tridiagonal matrix using divide and conquer algorithm.

Syntax

```
call sstevd (job, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
call dstevd (job, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric tridiagonal matrix T. In other words, the routine can compute the spectral factorization of T as: $T = Z\Lambda Z^T$.
Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i, and Z is the orthogonal matrix whose columns are the eigenvectors z_i. Thus,

$$Tz_i = \lambda_i z_i \quad \text{for } i = 1, 2, \ldots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

There is no complex analogue of this routine.

Input Parameters

job	`CHARACTER*1`. Must be `'N'` or `'V'`. If *job* =`'N'`, then only eigenvalues are computed. If *job* =`'V'`, then eigenvalues and eigenvectors are computed.
n	`INTEGER`. The order of the matrix T ($n \geq 0$).
d, e, work	`REAL` for `sstevd` `DOUBLE PRECISION` for `dstevd`. Arrays: *d*(*) contains the *n* diagonal elements of the tridiagonal matrix *T*. The dimension of *d* must be at least max(1, *n*).

e(*) contains the *n*-1 off-diagonal elements of *T*.
The dimension of *e* must be at least max(1, *n*). The *n*th element of this array is used as workspace.

work(*) is a workspace array.
The dimension of *work* must be at least *lwork*.

ldz — INTEGER. The leading dimension of the output array *z*. Constraints:
ldz ≥ 1 if *job* = 'N';
ldz ≥ max(1, *n*) if *job* = 'V'.

lwork — INTEGER. The dimension of the array *work*.
Constraints:
if *job* = 'N' or $n \le 1$, then *lwork* ≥ 1;
if *job* = 'V' and $n > 1$, then
lwork ≥ $2n^2+(3+2k)*n+1$, where *k* is the smallest integer which satisfies $2^k \ge n$.

iwork — INTEGER.
Workspace array, DIMENSION at least *liwork*.

liwork — INTEGER. The dimension of the array *iwork*.
Constraints:
if *job* = 'N' or $n \le 1$, then *liwork* ≥ 1;
if *job* = 'V' and $n > 1$, then *liwork* ≥ $5n+2$.

Output Parameters

d — On exit, if *info* = 0, contains the eigenvalues of the matrix *T* in ascending order.
See also *info*.

z — REAL for sstevd
DOUBLE PRECISION for dstevd
Array, DIMENSION (*ldz*, *).
The second dimension of *z* must be:
at least 1 if *job* = 'N';
at least max(1, *n*) if *job* = 'V'.

If *job* = 'V', then this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors
of *T*. If *job* = 'N', then *z* is not referenced.

e — On exit, this array is overwritten with intermediate results.

work(1) — On exit, if *lwork* > 0, then *work(1)* returns the required minimal size of *lwork*.

iwork(1) — On exit, if *liwork* > 0, then *iwork(1)* returns the required minimal size of *liwork*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.
If *info* = *-i*, the *i*th parameter had an illegal value.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $||E||_2 = O(\varepsilon) ||T||_2$, where ε is the machine precision.

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then

$$|\mu_i - \lambda_i| \le c(n)\varepsilon\, ||T||_2$$

where $c(n)$ is a modestly increasing function of n.

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$$\theta(z_i, w_i) \le c(n)\varepsilon\, ||T||_2 / \min_{i\ne j}|\lambda_i - \lambda_j|.$$

Thus the accuracy of a computed eigenvector depends on the gap between its eigenvalue and all the other eigenvalues.

?stevx

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

```
call sstevx ( jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z,
              ldz, work, iwork, ifail, info)
call dstevx ( jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z,
              ldz, work, iwork, ifail, info)
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *job* = `'N'`, then only eigenvalues are computed.
If *job* = `'V'`, then eigenvalues and eigenvectors are computed.

range — `CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`.
If *range* = `'A'`, the routine computes all eigenvalues.
If *range* = `'V'`, the routine computes eigenvalues λ_i in the half-open interval: $vl < \lambda_i \le vu$.
If *range* = `'I'`, the routine computes eigenvalues with indices *il* to *iu*.

n — `INTEGER`. The order of the matrix *A* ($n \ge 0$).

d, e, work — `REAL` for `sstevx`
`DOUBLE PRECISION` for `dstevx`.
Arrays:
d(*) contains the *n* diagonal elements of the tridiagonal matrix *A*.
The dimension of *d* must be at least max(1, *n*).

e(*) contains the *n*-1 subdiagonal elements of *A*.
The dimension of *e* must be at least max(1, *n*). The *n*th element of this array is used as workspace.

	`work(*)` is a workspace array. The dimension of *work* must be at least max(1, 5*n*).
vl, vu	`REAL` for `sstevx` `DOUBLE PRECISION` for `dstevx`. If *range* = `'V'`, the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: *vl*< *vu*. If *range* = `'A'` or `'I'`, *vl* and *vu* are not referenced.
il, iu	`INTEGER`. If *range* = `'I'`, the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: 1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0. If *range* = `'A'` or `'V'`, *il* and *iu* are not referenced.
abstol	`REAL` for `sstevx` `DOUBLE PRECISION` for `dstevx`. The absolute error tolerance to which each eigenvalue is required. See *Application notes* for details on error tolerance.
ldz	`INTEGER`. The leading dimensions of the output array *z*; *ldz* ≥ 1. If *jobz* = `'V'`, then *ldz* ≥ max(1, *n*).
iwork	`INTEGER`. Workspace array, `DIMENSION` at least max(1, 5*n*).

Output Parameters

m	`INTEGER`. The total number of eigenvalues found, 0 ≤ *m* ≤ *n*. If *range* = `'A'`, *m* = *n*, and if *range* = `'I'`, *m* = *iu*-*il*+1.
w, z	`REAL` for `sstevx` `DOUBLE PRECISION` for `dstevx`. Arrays: *w(*)*, `DIMENSION` at least max(1, *n*) . The first *m* elements of *w* contain the selected eigenvalues of the matrix *A* in ascending order.

z(ldz,)*. The second dimension of *z* must be at least max(1, *m*).
If *jobz* ='V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
If *jobz* ='N', then *z* is not referenced.
Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* ='V', the exact value of *m* is not known in advance and an upper bound must be used.

d, e — On exit, these arrays may be multiplied by a constant factor chosen to avoid overflow or underflow in computing the eigenvalues.

ifail — INTEGER.
Array, DIMENSION at least max(1, *n*).
If *jobz* ='V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.
If *jobz* ='N', then *ifail* is not referenced.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.
If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to
abstol + ε * max(|a|,|b|) , where ε is the machine precision. If *abstol* is less than or equal to zero, then $\varepsilon * \|A\|_1$ will be used in its place.
Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?stevr

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix using the Relatively Robust Representations.

Syntax

```
call sstevr ( jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z,
              ldz, isuppz, work, lwork, iwork, liwork, info)
call dstevr ( jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z,
              ldz, isuppz, work, lwork, iwork, liwork, info)
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix *T*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Whenever possible, `?stevr` calls `sstegr/dstegr` to compute the eigenspectrum using Relatively Robust Representations. `?stegr` computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various “good" LDL^T representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the i-th unreduced block of *T*,

(a) Compute $T - \sigma_i = L_i D_i L_i^T$, such that $L_i D_i L_i^T$ is a relatively robust representation;

(b) Compute the eigenvalues, λ_j, of $L_i D_i L_i^T$ to high relative accuracy by the *dqds* algorithm;

(c) If there is a cluster of close eigenvalues, "choose" σ_i close to the cluster, and go to step (a);

(d) Given the approximate eigenvalue λ_j of $L_i D_i L_i^T$, compute the corresponding eigenvector by forming a rank-revealing twisted factorization.

The desired accuracy of the output can be specified by the input parameter *`abstol`*.

The routine `?stevr` calls `sstegr/dstegr` when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard. `?stevr` calls `sstebz/dstebz` and `sstein/dstein` on non-IEEE machines and when partial spectrum requests are made.

Input Parameters

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then only eigenvalues are computed.
If *jobz* = `'V'`, then eigenvalues and eigenvectors are computed.

range — `CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`.
If *range* = `'A'`, the routine computes all eigenvalues.
If *range* = `'V'`, the routine computes eigenvalues λ_i in the half-open interval: $vl < \lambda_i \le vu$.
If *range* = `'I'`, the routine computes eigenvalues with indices *il* to *iu*.

For *range* = `'V'` or `'I'` and *iu*-*il* < *n*-1, `sstebz/dstebz` and `sstein/dstein` are called.

n — `INTEGER`. The order of the matrix *T* ($n \ge 0$).

d, e, work — `REAL` for `sstevr`
`DOUBLE PRECISION` for `dstevr`.
Arrays:
d(*) contains the *n* diagonal elements of the tridiagonal matrix *T*.
The dimension of *d* must be at least max(1, *n*).

e(*) contains the *n*-1 subdiagonal elements of *A*.
The dimension of *e* must be at least max(1, *n*). The *n*th element of this array is used as workspace.

work(*lwork*) is a workspace array.

vl, vu — `REAL` for `sstevr`
`DOUBLE PRECISION` for `dstevr`.
If *range* = `'V'`, the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: $vl < vu$.

If *range* = `'A'` or `'I'`, *vl* and *vu* are not referenced.

il, iu — `INTEGER`.
If *range* = `'I'`, the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: $1 \le il \le iu \le n$, if $n > 0$; *il*=1 and *iu*=0 if $n = 0$.

If *range* = `'A'` or `'V'`, *il* and *iu* are not referenced.

abstol — REAL for ssyevr
DOUBLE PRECISION for dsyevr.
The absolute error tolerance to which each eigenvalue/eigenvector is required. If *jobz* = 'V', the eigenvalues and eigenvectors output have residual norms bounded by *abstol*, and the dot products between different eigenvectors are bounded by *abstol*. If *abstol* < $n\varepsilon\|T\|_1$, then $n\varepsilon\|T\|_1$ will be used in its place, where ε is the machine precision. The eigenvalues are computed to an accuracy of $\varepsilon\|T\|_1$ irrespective of *abstol*. If high relative accuracy is important, set *abstol* to ?lamch('S').

ldz — INTEGER. The leading dimension of the output array *z*. Constraints:
ldz ≥ 1 if *jobz* = 'N';
ldz ≥ max(1, *n*) if *jobz* = 'V'.

lwork — INTEGER. The dimension of the array *work*.
Constraint: *lwork* ≥ max(1, 20*n*).

iwork — INTEGER.
Workspace array, DIMENSION (*liwork*).

liwork — INTEGER. The dimension of the array *iwork*,
lwork ≥ max(1, 10*n*).

Output Parameters

m — INTEGER. The total number of eigenvalues found,
0 ≤ *m* ≤ *n*. If *range* = 'A', *m* = *n*, and if *range* = 'I',
m = *iu*-*il*+1.

w, z — REAL for sstevr
DOUBLE PRECISION for dstevr.
Arrays:
w(*), DIMENSION at least max(1, *n*) .
The first *m* elements of *w* contain the selected eigenvalues of the matrix *T* in ascending order.

z(*ldz, **) . The second dimension of *z* must be at least max(1, *m*).
If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z* ; if *range* ='V', the exact value of *m* is not known in advance and an upper bound must be used.

d, e — On exit, these arrays may be multiplied by a constant factor chosen to avoid overflow or underflow in computing the eigenvalues.

isuppz — INTEGER.
Array, DIMENSION at least 2*max(1, *m*).

The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th eigenvector is nonzero only in elements *isuppz*(2*i*-1) through *isuppz*(2*i*).
Implemented only for *range* ='A' or 'I' and
iu-*il* = *n*-1.

work(1) — On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

iwork(1) — On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, an internal error has occurred.

Application Notes

Normal execution of the routine ?stegr may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

Nonsymmetric Eigenproblems

This section describes LAPACK driver routines used for solving nonsymmetric eigenproblems. See also computational routines that can be called to solve these problems. Table 4-12 lists routines described in more detail below.

Table 4-11 Driver Routines for Solving Nonsymmetric Eigenproblems

Routine Name	Operation performed
?gees	Computes the eigenvalues and Schur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Schur form.
?geesx	Computes the eigenvalues and Schur factorization of a general matrix, orders the factorization and computes reciprocal condition numbers.
?geev	Computes the eigenvalues and left and right eigenvectors of a general matrix.
?geevx	Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary matrix balancing, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.

?gees

Computes the eigenvalues and Schur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Schur form.

Syntax

```
call sgees ( jobvs, sort, select, n, a, lda, sdim, wr, wi, vs, ldvs,
             work, lwork, bwork, info)
call dgees ( jobvs, sort, select, n, a, lda, sdim, wr, wi, vs, ldvs,
             work, lwork, bwork, info)
call cgees ( jobvs, sort, select, n, a, lda, sdim, w, vs, ldvs,
             work, lwork, rwork, bwork, info)
call zgees ( jobvs, sort, select, n, a, lda, sdim, w, vs, ldvs,
             work, lwork, rwork, bwork, info)
```

Description

This routine computes for an n-by-n real/complex nonsymmetric matrix A, the eigenvalues, the real Schur form T, and, optionally, the matrix of Schur vectors Z. This gives the Schur factorization $A = ZTZ^H$.

Optionally, it also orders the eigenvalues on the diagonal of the real-Schur/Schur form so that selected eigenvalues are at the top left. The leading columns of Z then form an orthonormal basis for the invariant subspace corresponding to the selected eigenvalues.

A real matrix is in real-Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form

$$\begin{pmatrix} a & b \\ c & a \end{pmatrix}$$

where $b*c < 0$. The eigenvalues of such a block are $a \pm \sqrt{bc}$.

A complex matrix is in Schur form if it is upper triangular.

Input Parameters

jobvs
: `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobvs* = `'N'`, then Schur vectors are not computed.
If *jobvs* = `'V'`, then Schur vectors are computed.

sort
: `CHARACTER*1`. Must be `'N'` or `'S'`.
Specifies whether or not to order the eigenvalues on the diagonal of the Schur form.

If *sort* = `'N'`, then eigenvalues are not ordered.
If *sort* = `'S'`, eigenvalues are ordered (see *select*).

select
: `LOGICAL FUNCTION` of two `REAL` arguments for real flavors.
`LOGICAL FUNCTION` of one `COMPLEX` argument for complex flavors.

select must be declared `EXTERNAL` in the calling subroutine.
If *sort* = `'S'`, *select* is used to select eigenvalues to sort to the top left of the Schur form.
If *sort* = `'N'`, *select* is not referenced.
For real flavors:
An eigenvalue *wr*(j)+ $\sqrt{-1}$ **wi*(j) is selected if *select*(*wr*(j), *wi*(j)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then

both complex eigenvalues are selected. Note that a selected complex eigenvalue may no longer satisfy *select*(*wr*(j), *wi*(j)) = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case *info* may be set to *n*+2 (see *info* below).
For complex flavors:
An eigenvalue *w*(j) is selected if *select*(*w*(j)) is true.

n — INTEGER. The order of the matrix A ($n \ge 0$).

a, work — REAL for sgees
DOUBLE PRECISION for dgees
COMPLEX for cgees
DOUBLE COMPLEX for zgees.
Arrays:
a(*lda*,*) is an array containing the *n*-by-*n* matrix A.
The second dimension of *a* must be at least max(1, *n*).

work(*lwork*) is a workspace array.

lda — INTEGER. The first dimension of the array *a*.
Must be at least max(1, *n*) .

ldvs — INTEGER. The leading dimension of the output array *vs*. Constraints:
$ldvs \ge 1$;
$ldvs \ge \max(1, n)$ if *jobvs* ='V'.

lwork — INTEGER. The dimension of the array *work*.
Constraint:
$lwork \ge \max(1, 3n)$ for real flavors;
$lwork \ge \max(1, 2n)$ for complex flavors.

rwork — REAL for cgees
DOUBLE PRECISION for zgees
Workspace array, DIMENSION at least max(1, *n*). Used in complex flavors only.

bwork — LOGICAL.
Workspace array, DIMENSION at least max(1, *n*). Not referenced if *sort* ='N'.

Output Parameters

a — On exit, this array is overwritten by the real-Schur/Schur form T .

sdim — `INTEGER`.
If *sort* ='N', *sdim*= 0.
If *sort* ='S', *sdim* is equal to the number of eigenvalues (after sorting) for which *select* is true.
Note that for real flavors complex conjugate pairs for which *select* is true for either eigenvalue count as 2.

wr, wi — `REAL` for `sgees`
`DOUBLE PRECISION` for `dgees`
Arrays, `DIMENSION` at least max (1, *n*) each.
Contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output real-Schur form *T*. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w — `COMPLEX` for `cgees`
`DOUBLE COMPLEX` for `zgees`.
Array, `DIMENSION` at least max(1,*n*).
Contains the computed eigenvalues. The eigenvalues are stored in the same order as they appear on the diagonal of the output Schur form *T*.

vs — `REAL` for `sgees`
`DOUBLE PRECISION` for `dgees`
`COMPLEX` for `cgees`
`DOUBLE COMPLEX` for `zgees`.
Array *vs*(*ldvs*,*); the second dimension of *vs* must be at least max(1, *n*).

If *jobvs* ='V', *vs* contains the orthogonal/unitary matrix *Z* of Schur vectors.
If *jobvs* ='N', *vs* is not referenced.

work(1) — On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info — `INTEGER`.
If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

If *info* = *i*, and
$i \le n$:

the *QR* algorithm failed to compute all the eigenvalues; elements 1:`ilo`-1 and `i`+1:`n` of `wr` and `wi` (for real flavors) or `w` (for complex flavors) contain those eigenvalues which have converged; if `jobvs` = 'V', `vs` contains the matrix which reduces *A* to its partially converged Schur form;

`i` = `n`+1 :

the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned);

`i` = `n`+2 :

after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy `select` = `.TRUE.`. This could also be caused by underflow due to scaling.

Application Notes

If you are in doubt how much workspace to supply for the array `work`, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

?geesx

Computes the eigenvalues and Schur factorization of a general matrix, orders the factorization and computes reciprocal condition numbers.

Syntax

```
call sgeesx(jobvs, sort, select, sense, n, a, lda, sdim, wr, wi, vs,
       ldvs, rconde, rcondv, work, lwork, iwork, liwork, bwork, info)
call dgeesx(jobvs, sort, select, sense, n, a, lda, sdim, wr, wi, vs,
       ldvs, rconde, rcondv, work, lwork, iwork, liwork, bwork, info)
call cgeesx(jobvs, sort, select, sense, n, a, lda, sdim, w, vs,
             ldvs, rconde, rcondv, work, lwork, rwork, bwork, info)
call zgeesx(jobvs, sort, select, sense, n, a, lda, sdim, w, vs,
             ldvs, rconde, rcondv, work, lwork, rwork, bwork, info)
```

Description

This routine computes for an n-by-n real/complex nonsymmetric matrix A, the eigenvalues, the real-Schur/Schur form T, and, optionally, the matrix of Schur vectors Z. This gives the Schur factorization $A = Z T Z^H$.

Optionally, it also orders the eigenvalues on the diagonal of the real-Schur/Schur form so that selected eigenvalues are at the top left; computes a reciprocal condition number for the average of the selected eigenvalues (`rconde`); and computes a reciprocal condition number for the right invariant subspace corresponding to the selected eigenvalues (`rcondv`). The leading columns of Z form an orthonormal basis for this invariant subspace.

For further explanation of the reciprocal condition numbers `rconde` and `rcondv`, see [LUG], Section 4.10 (where these quantities are called *s* and *sep* respectively).

A real matrix is in real-Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form

$$\begin{pmatrix} a & b \\ c & a \end{pmatrix}$$

where $b*c < 0$. The eigenvalues of such a block are $a \pm \sqrt{bc}$.

A complex matrix is in Schur form if it is upper triangular.

Input Parameters

jobvs CHARACTER*1. Must be 'N' or 'V'.
If *jobvs* ='N', then Schur vectors are not computed.
If *jobvs* ='V', then Schur vectors are computed.

sort CHARACTER*1. Must be 'N' or 'S'.
Specifies whether or not to order the eigenvalues on the diagonal of the Schur form.

If *sort* ='N', then eigenvalues are not ordered.
If *sort* ='S', eigenvalues are ordered (see *select*).

select LOGICAL FUNCTION of two REAL arguments for real flavors.
LOGICAL FUNCTION of one COMPLEX argument for complex flavors.

select must be declared EXTERNAL in the calling subroutine.
If *sort* ='S', *select* is used to select eigenvalues to sort to the top left of the Schur form.
If *sort* ='N', *select* is not referenced.
For real flavors:
An eigenvalue *wr*(j)+ $\sqrt{-1}$ **wi*(j) is selected if *select*(*wr*(j), *wi*(j)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected. Note that a selected complex eigenvalue may no longer satisfy *select*(*wr*(j), *wi*(j)) = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case *info* may be set to *n*+2 (see *info* below).
For complex flavors:
An eigenvalue *w*(j) is selected if *select*(*w*(j)) is true.

sense CHARACTER*1. Must be 'N', 'E', 'V', or 'B'.
Determines which reciprocal condition number are computed.

If *sense* ='N', none are computed;
If *sense* ='E', computed for average of selected eigenvalues only;
If *sense* ='V', computed for selected right invariant subspace only;
If *sense* ='B', computed for both.

If *sense* is 'E', 'V', or 'B', then *sort* must equal 'S'.

n INTEGER. The order of the matrix *A* ($n \ge 0$).

a, work	REAL for sgeesx DOUBLE PRECISION for dgeesx COMPLEX for cgeesx DOUBLE COMPLEX for zgeesx. Arrays: a(*lda*,*) is an array containing the *n*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work*(*lwork*) is a workspace array.
lda	INTEGER. The first dimension of the array *a*. Must be at least max(1, *n*) .
ldvs	INTEGER. The leading dimension of the output array *vs*. Constraints: *ldvs* ≥ 1 ; *ldvs* ≥ max(1, *n*) if *jobvs* ='V'.
lwork	INTEGER. The dimension of the array *work*. Constraint: *lwork* ≥ max(1, 3*n*) for real flavors; *lwork* ≥ max(1, 2*n*) for complex flavors. Also, if *sense* = 'E', 'V', or 'B', then *lwork* ≥ *n*+2**sdim**(*n*-*sdim*) for real flavors; *lwork* ≥ 2**sdim**(*n*-*sdim*) for complex flavors; where *sdim* is the number of selected eigenvalues computed by this routine. Note that 2**sdim**(*n*-*sdim*) ≤ *n***n*/2 . For good performance, *lwork* must generally be larger.
iwork	INTEGER. Workspace array, DIMENSION (*liwork*). Used in real flavors only. Not referenced if *sense* = 'N' or 'E'.
liwork	INTEGER. The dimension of the array *iwork*. Used in real flavors only. Constraint: *liwork* ≥ 1; if *sense* = 'V' or 'B', *liwork* ≥ *sdim**(*n*-*sdim*).
rwork	REAL for cgeesx DOUBLE PRECISION for zgeesx Workspace array, DIMENSION at least max(1, *n*). Used in complex flavors only.

bwork LOGICAL.
Workspace array, DIMENSION at least max(1, *n*). Not referenced if *sort* ='N'.

Output Parameters

a On exit, this array is overwritten by the real-Schur/Schur form *T* .

sdim INTEGER.
If *sort* ='N', *sdim*= 0.
If *sort* ='S', *sdim* is equal to the number of eigenvalues (after sorting) for which *select* is true.
Note that for real flavors complex conjugate pairs for which *select* is true for either eigenvalue count as 2.

wr, wi REAL for sgeesx
DOUBLE PRECISION for dgeesx
Arrays, DIMENSION at least max (1, *n*) each.
Contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output real-Schur form *T*. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w COMPLEX for cgeesx
DOUBLE COMPLEX for zgeesx.
Array, DIMENSION at least max(1,*n*).
Contains the computed eigenvalues. The eigenvalues are stored in the same order as they appear on the diagonal of the output Schur form *T*.

vs REAL for sgeesx
DOUBLE PRECISION for dgeesx
COMPLEX for cgeesx
DOUBLE COMPLEX for zgeesx.
Array *vs*(*ldvs*,*); the second dimension of *vs* must be at least max(1, *n*).

If *jobvs* ='V', *vs* contains the orthogonal/unitary matrix *Z* of Schur vectors.
If *jobvs* ='N', *vs* is not referenced.

rconde, rcondv REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
If *sense* = 'E' or 'B', *rconde* contains the reciprocal condition number for the average of the selected eigenvalues. If *sense* = 'N' or 'V', *rconde* is not referenced.

If *sense* = 'V' or 'B', *rcondv* contains the reciprocal condition number for the selected right invariant subspace. If *sense* = 'N' or 'E', *rcondv* is not referenced.

work(1) On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info INTEGER.
If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

If *info* = *i*, and
$i \le n$:

the *QR* algorithm failed to compute all the eigenvalues; elements 1:*ilo*-1 and *i*+1:*n* of *wr* and *wi* (for real flavors) or *w* (for complex flavors) contain those eigenvalues which have converged; if *jobvs* ='V', *vs* contains the transformation which reduces *A* to its partially converged Schur form;

$i = n+1$:

the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned);

$i = n+2$:

after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy *select* = .TRUE.. This could also be caused by underflow due to scaling.

Application Notes

If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

?geev

Computes the eigenvalues and left and right eigenvectors of a general matrix.

Syntax

```
call sgeev ( jobvl, jobvr, n, a, lda, wr, wi, vl, ldvl, vr, ldvr,
             work, lwork, info)
call dgeev ( jobvl, jobvr, n, a, lda, wr, wi, vl, ldvl, vr, ldvr,
             work, lwork, info)
call cgeev ( jobvl, jobvr, n, a, lda, w, vl, ldvl, vr, ldvr, work,
             lwork, rwork, info)
call zgeev ( jobvl, jobvr, n, a, lda, w, vl, ldvl, vr, ldvr, work,
             lwork, rwork, info)
```

Description

This routine computes for an *n*-by-*n* real/complex nonsymmetric matrix *A*, the eigenvalues and, optionally, the left and/or right eigenvectors. The right eigenvector *v*(j) of *A* satisfies

$$A * v(j) = \lambda(j) * v(j)$$

where λ(j) is its eigenvalue.

The left eigenvector *u*(j) of *A* satisfies

$$u(j)^H * A = \lambda(j) * u(j)^H$$

where $u(j)^H$ denotes the conjugate transpose of *u*(j).
The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Input Parameters

jobvl — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobvl* = `'N'`, then left eigenvectors of *A* are not computed.
If *jobvl* = `'V'`, then left eigenvectors of *A* are computed.

jobvr — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobvr* = `'N'`, then right eigenvectors of *A* are not computed.
If *jobvr* = `'V'`, then right eigenvectors of *A* are computed.

n	`INTEGER`. The order of the matrix *A* ($n \geq 0$).
a, work	`REAL` for `sgeev` `DOUBLE PRECISION` for `dgeev` `COMPLEX` for `cgeev` `DOUBLE COMPLEX` for `zgeev`. Arrays: *a*(*lda*,*) is an array containing the *n*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work*(*lwork*) is a workspace array.
lda	`INTEGER`. The first dimension of the array *a*. Must be at least max(1, *n*) .
ldvl, ldvr	`INTEGER`. The leading dimensions of the output arrays *vl* and *vr*, respectively. Constraints: *ldvl* ≥ 1 ; *ldvr* ≥ 1. If *jobvl* = 'V', *ldvl* ≥ max(1, *n*) ; If *jobvr* = 'V', *ldvr* ≥ max(1, *n*).
lwork	`INTEGER`. The dimension of the array *work*. Constraint: *lwork* ≥ max(1, 3*n*) , and if *jobvl* = 'V' or *jobvr* = 'V', *lwork* ≥ max(1, 4*n*) (for real flavors); *lwork* ≥ max(1, 2*n*) (for complex flavors). For good performance, *lwork* must generally be larger.
rwork	`REAL` for `cgeev` `DOUBLE PRECISION` for `zgeev` Workspace array, `DIMENSION` at least max(1, 2*n*). Used in complex flavors only.

Output Parameters

a	On exit, this array is overwritten by intermediate results.
wr, wi	`REAL` for `sgeev` `DOUBLE PRECISION` for `dgeev` Arrays, `DIMENSION` at least max (1, *n*) each. Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w	COMPLEX for cgeev DOUBLE COMPLEX for zgeev. Array, DIMENSION at least max(1,*n*). Contains the computed eigenvalues.
vl, vr	REAL for sgeev DOUBLE PRECISION for dgeev COMPLEX for cgeev DOUBLE COMPLEX for zgeev. Arrays: *vl(ldvl,*)*; the second dimension of *vl* must be at least max(1, *n*). If *jobvl* ='V', the left eigenvectors *u*(j) are stored one after another in the columns of *vl*, in the same order as their eigenvalues. If *jobvl* ='N', *vl* is not referenced. *For real flavors:* If the j-th eigenvalue is real, then *u*(j) = *vl*(:,j), the j-th column of *vl*. If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then *u*(j) = *vl*(:,j) + *i***vl*(:,j+1) and *u*(j+1) = *vl*(:,j) - *i***vl*(:,j+1), where $i=\sqrt{-1}$. *For complex flavors:* *u*(j) = *vl*(:,j), the j-th column of *vl*. *vr(ldvr,*)*; the second dimension of *vr* must be at least max(1, *n*). If *jobvr* ='V', the right eigenvectors *v*(j) are stored one after another in the columns of *vr*, in the same order as their eigenvalues. If *jobvr* ='N', *vr* is not referenced. *For real flavors:* If the j-th eigenvalue is real, then *v*(j) = *vr*(:,j), the j-th column of *vr*. If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then *v*(j) = *vr*(:,j) + *i***vr*(:,j+1) and *v*(j+1) = *vr*(:,j) - *i***vr*(:,j+1), where $i=\sqrt{-1}$. *For complex flavors:* *v*(j) = *vr*(:,j), the j-th column of *vr*.
work(1)	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i*, the *QR* algorithm failed to compute all the eigenvalues, and no

eigenvectors have been computed; elements *i*+1:*n* of *wr* and *wi* (for real flavors) or *w* (for complex flavors) contain those eigenvalues which have converged.

Application Notes

If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

?geevx

Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary matrix balancing, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.

Syntax

```
call sgeevx ( balanc, jobvl, jobvr, sense, n, a, lda, wr, wi, vl,
              ldvl, vr, ldvr, ilo, ihi, scale, abnrm, rconde,
               rcondv, work, lwork, iwork, info)
call dgeevx ( balanc, jobvl, jobvr, sense, n, a, lda, wr, wi, vl,
              ldvl, vr, ldvr, ilo, ihi, scale, abnrm, rconde,
               rcondv, work, lwork, iwork, info)
call cgeevx ( balanc, jobvl, jobvr, sense, n, a, lda, w, vl, ldvl,
              vr, ldvr, ilo, ihi, scale, abnrm, rconde, rcondv,
               work, lwork, rwork, info)
call zgeevx ( balanc, jobvl, jobvr, sense, n, a, lda, w, vl, ldvl,
              vr, ldvr, ilo, ihi, scale, abnrm, rconde, rcondv,
               work, lwork, rwork, info)
```

Description

This routine computes for an *n*-by-*n* real/complex nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (*ilo*, *ihi*, *scale*, and *abnrm*), reciprocal condition numbers for the eigenvalues (*rconde*), and reciprocal condition numbers for the right eigenvectors (*rcondv*).

The right eigenvector $v(j)$ of A satisfies

$$A*v(j) = \lambda(j)*v(j)$$

where $\lambda(j)$ is its eigenvalue.

The left eigenvector $u(j)$ of A satisfies

$$u(j)^H*A = \lambda(j)*u(j)^H$$

where $u(j)^H$ denotes the conjugate transpose of u(j).
The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Balancing a matrix means permuting the rows and columns to make it more nearly upper triangular, and applying a diagonal similarity transformation $D A D^{-1}$, where D is a diagonal matrix, to make its rows and columns closer in norm and the condition numbers of its eigenvalues and eigenvectors smaller. The computed reciprocal condition numbers correspond to the balanced matrix.
Permuting rows and columns will not change the condition numbers in exact arithmetic) but diagonal scaling will. For further explanation of balancing, see [LUG], Section 4.10.

Input Parameters

balanc — `CHARACTER*1`. Must be `'N'`, `'P'`, `'S'`, or `'B'`.
Indicates how the input matrix should be diagonally scaled and/or permuted to improve the conditioning of its eigenvalues.

If *balanc* = `'N'`, do not diagonally scale or permute;
If *balanc* = `'P'`, perform permutations to make the matrix more nearly upper triangular. Do not diagonally scale;
If *balanc* = `'S'`, Diagonally scale the matrix, i.e. replace A by $D A D^{-1}$, where D is a diagonal matrix chosen to make the rows and columns of A more equal in norm. Do not permute;
If *balanc* = `'B'`, both diagonally scale and permute A.

Computed reciprocal condition numbers will be for the matrix after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.

jobvl — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobvl* = `'N'`, left eigenvectors of A are not computed;
If *jobvl* = `'V'`, left eigenvectors of A are computed.
If *sense* = `'E'` or `'B'`, then *jobvl* must be `'V'`.

jobvr — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobvr* = `'N'`, right eigenvectors of A are not computed;
If *jobvr* = `'V'`, right eigenvectors of A are computed.
If *sense* = `'E'` or `'B'`, then *jobvr* must be `'V'`.

sense — `CHARACTER*1`. Must be `'N'`, `'E'`, `'V'`, or `'B'`.
Determines which reciprocal condition number are computed.

If *sense* = 'N', none are computed;
If *sense* = 'E', computed for eigenvalues only;
If *sense* = 'V', computed for right eigenvectors only;
If *sense* = 'B', computed for eigenvalues and right eigenvectors.

If *sense* is 'E' or 'B', both left and right eigenvectors must also be computed (*jobvl* = 'V' and *jobvr* = 'V').

n — INTEGER. The order of the matrix A ($n \geq 0$).

a, work — REAL for sgeevx
DOUBLE PRECISION for dgeevx
COMPLEX for cgeevx
DOUBLE COMPLEX for zgeevx.
Arrays:
a(*lda*,*) is an array containing the *n*-by-*n* matrix A.
The second dimension of *a* must be at least max(1, *n*).

work(*lwork*) is a workspace array.

lda — INTEGER. The first dimension of the array *a*.
Must be at least max(1, *n*) .

ldvl, ldvr — INTEGER. The leading dimensions of the output arrays *vl* and *vr*, respectively.
Constraints:
$ldvl \geq 1$; $ldvr \geq 1$.
If *jobvl* = 'V', $ldvl \geq \max(1, n)$;
If *jobvr* = 'V', $ldvr \geq \max(1, n)$.

lwork — INTEGER. The dimension of the array *work*.
For real flavors:
If *sense* = 'N' or 'E', $lwork \geq \max(1, 2n)$, and
if *jobvl* = 'V' or *jobvr* = 'V', $lwork \geq 3n$;
If *sense* = 'V' or 'B', $lwork \geq n(n+6)$.
For good performance, *lwork* must generally be larger.

For complex flavors:
If *sense* = 'N' or 'E', $lwork \geq \max(1, 2n)$;
If *sense* = 'V' or 'B', $lwork \geq n^2+2n$.
For good performance, *lwork* must generally be larger.

rwork — REAL for cgeevx
DOUBLE PRECISION for zgeevx
Workspace array, DIMENSION at least max(1, 2*n*). Used in complex flavors only.

iwork — `INTEGER`.
Workspace array, `DIMENSION` at least max(1, 2*n*-2). Used in real flavors only. Not referenced if *sense* = 'N' or 'E'.

Output Parameters

a — On exit, this array is overwritten. If *jobvl* ='V' or *jobvr* ='V', it contains the real-Schur/Schur form of the balanced version of the input matrix *A*.

wr, wi — `REAL` for `sgeevx`
`DOUBLE PRECISION` for `dgeevx`
Arrays, `DIMENSION` at least max (1, *n*) each.
Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w — `COMPLEX` for `cgeevx`
`DOUBLE COMPLEX` for `zgeevx`.
Array, `DIMENSION` at least max(1,*n*).
Contains the computed eigenvalues.

vl, vr — `REAL` for `sgeevx`
`DOUBLE PRECISION` for `dgeevx`
`COMPLEX` for `cgeevx`
`DOUBLE COMPLEX` for `zgeevx`.
Arrays:
vl(*ldvl*,*); the second dimension of *vl* must be at least max(1, *n*).

If *jobvl* ='V', the left eigenvectors *u*(j) are stored one after another in the columns of *vl*, in the same order as their eigenvalues. If *jobvl* ='N', *vl* is not referenced.
For real flavors:
If the j-th eigenvalue is real, then *u*(j) = *vl*(:,j), the j-th column of *vl*. If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then *u*(j) = *vl*(:,j) + *i***vl*(:,j+1) and *u*(j+1) = *vl*(:,j) - *i***vl*(:,j+1), where $i=\sqrt{-1}$.

For complex flavors:
u(j) = *vl*(:,j), the j-th column of *vl*.

vr(*ldvr*,*); the second dimension of *vr* must be at least max(1, *n*).

If *jobvr* ='V', the right eigenvectors *v*(j) are stored one after another in the columns of *vr*, in the same order as their eigenvalues. If *jobvr* ='N', *vr* is not referenced.
For real flavors:

If the j-th eigenvalue is real, then *v*(j) = `vr`(:,j), the j-th column of `vr`. If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then *v*(j) = `vr`(:,j) + *i**`vr`(:,j+1) and *v*(j+1) = `vr`(:,j) - *i**`vr`(:,j+1), where $i = \sqrt{-1}$.

For complex flavors:
v(j) = `vr`(:,j), the j-th column of `vr`.

ilo, ihi — `INTEGER`.
`ilo` and `ihi` are integer values determined when *A* was balanced.
The balanced *A*(i,j) = 0 if i > j and j = 1,..., `ilo`-1 or
i = `ihi`+1,..., *n*.
If `balanc` = `'N'` or `'S'`, `ilo` = 1 and `ihi` = *n*.

scale — `REAL` for single-precision flavors
`DOUBLE PRECISION` for double-precision flavors.
Array, `DIMENSION` at least max(1, *n*) .
Details of the permutations and scaling factors applied when balancing *A*. If *P*(j) is the index of the row and column interchanged with row and column j, and *D*(j) is the scaling factor applied to row and column j, then

`scale`(j) = *P*(j), for j = 1,...,`ilo`-1

= *D*(j), for j = `ilo`,...,`ihi`

= *P*(j) for j = `ihi`+1,...,*n*.

The order in which the interchanges are made is *n* to `ihi`+1, then 1 to `ilo`-1.

abnrm — `REAL` for single-precision flavors
`DOUBLE PRECISION` for double-precision flavors.

The one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).

rconde, rcondv — `REAL` for single precision flavors
`DOUBLE PRECISION` for double precision flavors.
Arrays, `DIMENSION` at least max(1, *n*) each.
`rconde`(j) is the reciprocal condition number of the j-th eigenvalue.

`rcondv`(j) is the reciprocal condition number of the j-th right eigenvector.

work(1) — On exit, if `info` = 0, then `work(1)` returns the required minimal size of `lwork`.

info — `INTEGER`.
If `info` = 0, the execution is successful.
If `info` = `-i`, the `i`th parameter had an illegal value.

If *info* = *i*, the *QR* algorithm failed to compute all the eigenvalues, and no eigenvectors or condition numbers have been computed; elements 1:*ilo*-1 and *i*+1:*n* of *wr* and *wi* (for real flavors) or *w* (for complex flavors) contain eigenvalues which have converged.

Application Notes

If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

Singular Value Decomposition

This section describes LAPACK driver routines used for solving singular value problems. See also computational routines that can be called to solve these problems.
Table 4-12 lists routines described in more detail below.

Table 4-12 Driver Routines for Singular Value Decomposition

Routine Name	Operation performed
?gesvd	Computes the singular value decomposition of a general rectangular matrix.
?gesdd	Computes the singular value decomposition of a general rectangular matrix using a divide and conquer method.
?ggsvd	Computes the generalized singular value decomposition of a pair of general rectangular matrices.

?gesvd

Computes the singular value decomposition of a general rectangular matrix.

Syntax

```
call sgesvd ( jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt,
              work, lwork, info)
call dgesvd ( jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt,
              work, lwork, info)
call cgesvd ( jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt,
              work, lwork, rwork, info)
call zgesvd ( jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt,
              work, lwork, rwork, info)
```

Description

This routine computes the singular value decomposition (SVD) of a real/complex m-by-n matrix A, optionally computing the left and/or right singular vectors. The SVD is written

$$A = U \Sigma V^H$$

where Σ is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal/unitary matrix, and V is an n-by-n orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of A; they are real and non-negative, and are returned in

descending order. The first min(*m*,*n*) columns of *U* and *V* are the left and right singular vectors of *A*.
Note that the routine returns V^H, not *V*.

Input Parameters

jobu CHARACTER*1. Must be 'A', 'S', 'O', or 'N'.
Specifies options for computing all or part of the matrix *U*.

If *jobu* ='A', all *m* columns of *U* are returned in the array *u*;
if *jobu* ='S', the first min(*m*,*n*) columns of *U* (the left singular vectors) are returned in the array *u*;
if *jobu* ='O', the first min(*m*,*n*) columns of *U* (the left singular vectors) are overwritten on the array *a*;
if *jobu* ='N', no columns of *U* (no left singular vectors) are computed.

jobvt CHARACTER*1. Must be 'A', 'S', 'O', or 'N'.
Specifies options for computing all or part of the matrix V^H.

If *jobvt* ='A', all *n* rows of V^H are returned in the array *vt*;
if *jobvt* ='S', the first min(*m*,*n*) rows of V^H (the right singular vectors) are returned in the array *vt*;
if *jobvt* ='O', the first min(*m*,*n*) rows of V^H (the right singular vectors) are overwritten on the array *a*;
if *jobvt* ='N', no rows of V^H (no right singular vectors) are computed.

jobvt and *jobu* cannot both be 'O'.

m INTEGER. The number of rows of the matrix *A* ($m \geq 0$).

n INTEGER. The number of columns in *A* ($n \geq 0$).

a, work REAL for sgesvd
DOUBLE PRECISION for dgesvd
COMPLEX for cgesvd
DOUBLE COMPLEX for zgesvd.
Arrays:
a(*lda*,*) is an array containing the *m*-by-*n* matrix *A*.
The second dimension of *a* must be at least max(1, *n*).

work(*lwork*) is a workspace array.

lda — INTEGER. The first dimension of the array *a*.
Must be at least max(1, *m*) .

ldu, ldvt — INTEGER. The leading dimensions of the output arrays *u* and *vt*, respectively.
Constraints:
$ldu \ge 1$; $ldvt \ge 1$.
If *jobu* ='S' or 'A', $ldu \ge m$;
If *jobvt* ='A', $ldvt \ge n$;
If *jobvt* ='S', $ldvt \ge \min(m, n)$.

lwork — INTEGER. The dimension of the array *work*; $lwork \ge 1$.
Constraints:
$lwork \ge \max(3*\min(m,n)+\max(m,n), 5*\min(m,n))$ (for real flavors);
$lwork \ge 2*\min(m,n)+\max(m,n)$ (for complex flavors).
For good performance, *lwork* must generally be larger.

rwork — REAL for cgesvd
DOUBLE PRECISION for zgesvd
Workspace array, DIMENSION at least
max(1, 5*min(*m*,*n*)). Used in complex flavors only.

Output Parameters

a — On exit,
If *jobu* ='O', *a* is overwritten with the first min(*m*,*n*) columns of *U* (the left singular vectors, stored columnwise);
If *jobvt* ='O', *a* is overwritten with the first min(*m*,*n*) rows of V^H (the right singular vectors, stored rowwise);
If *jobu* ≠'O' and *jobvt* ≠'O', the contents of *a* are destroyed.

s — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Array, DIMENSION at least max(1, min(*m*,*n*)).
Contains the singular values of *A* sorted so that
$s(i) \ge s(i+1)$.

u, vt — REAL for sgesvd
DOUBLE PRECISION for dgesvd
COMPLEX for cgesvd
DOUBLE COMPLEX for zgesvd.
Arrays:
u(*ldu*,*); the second dimension of *u* must be at least max(1, *m*) if *jobu* ='A', and at least max(1, min(*m*,*n*)) if *jobu* ='S'.

If *jobu* = 'A', *u* contains the *m*-by-*m* orthogonal/unitary matrix *U*.
If *jobu* = 'S', *u* contains the first min(*m*,*n*) columns of *U* (the left singular vectors, stored columnwise).
If *jobu* = 'N' or 'O', *u* is not referenced.

vt(*ldvt*,*); the second dimension of *vt* must be at least max(1, *n*).

If *jobvt* = 'A', *vt* contains the *n*-by-*n* orthogonal/unitary matrix V^H.
If *jobvt* = 'S', *vt* contains the first min(*m*,*n*) rows of V^H (the right singular vectors, stored rowwise).
If *jobvt* = 'N' or 'O', *vt* is not referenced.

work — On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
For real flavors:
If *info* > 0, *work*(2:min(*m*,*n*)) contains the unconverged superdiagonal elements of an upper bidiagonal matrix *B* whose diagonal is in *s* (not necessarily sorted). *B* satisfies *A* = *u* * *B* * *vt*, so it has the same singular values as *A*, and singular vectors related by *u* and *vt*.

rwork — On exit (for complex flavors), if *info* > 0, *rwork*(1:min(*m*,*n*)-1) contains the unconverged superdiagonal elements of an upper bidiagonal matrix *B* whose diagonal is in *s* (not necessarily sorted). *B* satisfies *A* = *u* * *B* * *vt*, so it has the same singular values as *A*, and singular vectors related by *u* and *vt*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, then if ?bdsqr did not converge, *i* specifies how many superdiagonals of the intermediate bidiagonal form *B* did not converge to zero.

Application Notes

If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

?gesdd

Computes the singular value decomposition of a general rectangular matrix using a divide and conquer method.

Syntax

```
call sgesdd ( jobz, m, n, a, lda, s, u, ldu, vt, ldvt,
              work, lwork, iwork, info)
call dgesdd ( jobz, m, n, a, lda, s, u, ldu, vt, ldvt,
              work, lwork, iwork, info)
call cgesdd ( jobz, m, n, a, lda, s, u, ldu, vt, ldvt,
              work, lwork, rwork, iwork, info)
call zgesdd ( jobz, m, n, a, lda, s, u, ldu, vt, ldvt,
              work, lwork, rwork, iwork, info)
```

Description

This routine computes the singular value decomposition (SVD) of a real/complex *m*-by-*n* matrix *A*, optionally computing the left and/or right singular vectors. If singular vectors are desired, it uses a divide and conquer algorithm.
The SVD is written

$$A = U \Sigma V^H$$

where Σ is an *m*-by-*n* matrix which is zero except for its min(*m*,*n*) diagonal elements, *U* is an *m*-by-*m* orthogonal/unitary matrix, and *V* is an *n*-by-*n* orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of *A*; they are real and non-negative, and are returned in descending order. The first min(*m*,*n*) columns of *U* and *V* are the left and right singular vectors of *A*.
Note that the routine returns V^H, not *V*.

Input Parameters

jobz — `CHARACTER*1`. Must be `'A'`, `'S'`, `'O'`, or `'N'`.
Specifies options for computing all or part of the matrix *U*.

If *jobz* = 'A', all *m* columns of U and all *n* rows of V^T are returned in the arrays *u* and *vt*;
if *jobz* = 'S', the first min(*m*,*n*) columns of U and the first min(*m*,*n*) rows of V^T are returned in the arrays *u* and *vt*;
if *jobz* = 'O', then

if $m \geq n$, the first *n* columns of U are overwritten on the array *a* and all rows of V^T are returned in the array *vt*;
if $m < n$, all columns of U are returned in the array *u* and the first *m* rows of V^T are overwritten in the array *vt*;

if *jobz* = 'N', no columns of U or rows of V^T are computed.

m — INTEGER. The number of rows of the matrix A ($m \geq 0$).

n — INTEGER. The number of columns in A ($n \geq 0$).

a, work — REAL for sgesdd
DOUBLE PRECISION for dgesdd
COMPLEX for cgesdd
DOUBLE COMPLEX for zgesdd.
Arrays:
a(*lda*,*) is an array containing the *m*-by-*n* matrix A.
The second dimension of *a* must be at least max(1, *n*).

work(*lwork*) is a workspace array.

lda — INTEGER. The first dimension of the array *a*.
Must be at least max(1, *m*) .

ldu, ldvt — INTEGER. The leading dimensions of the output arrays *u* and *vt*, respectively.
Constraints:
$ldu \geq 1$; $ldvt \geq 1$.
If *jobz* = 'S' or 'A', or *jobz* = 'O' and $m < n$,
then $ldu \geq m$;
If *jobz* = 'A' or *jobz* = 'O' and $m \geq n$,
then $ldvt \geq n$;
If *jobz* = 'S', $ldvt \geq \min(m, n)$.

lwork — INTEGER. The dimension of the array *work*; $lwork \geq 1$.
See *Application Notes* for the suggested value of *lwork*.

rwork — REAL for cgesdd
DOUBLE PRECISION for zgesdd
Workspace array, DIMENSION at least

	max(1, 5*min(*m*,*n*)) if *jobz* = 'N'. Otherwise, the dimension of *rwork* must be at least $5*(\min(m,n))^2 + 7*\min(m,n)$. This array is used in complex flavors only.
iwork	INTEGER. Workspace array, DIMENSION at least max(1, 8*min(*m*,*n*)).

Output Parameters

a	On exit: If *jobz* = 'O', then if $m \ge n$, *a* is overwritten with the first *n* columns of *U* (the left singular vectors, stored columnwise). If $m < n$, *a* is overwritten with the first *m* rows of V^T (the right singular vectors, stored rowwise); If *jobz* ≠ 'O', the contents of *a* are destroyed.
s	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least max(1, min(*m*,*n*)). Contains the singular values of *A* sorted so that $s(i) \ge s(i+1)$.
u, vt	REAL for sgesdd DOUBLE PRECISION for dgesdd COMPLEX for cgesdd DOUBLE COMPLEX for zgesdd. Arrays: *u*(*ldu*,*); the second dimension of *u* must be at least max(1, *m*) if *jobz* = 'A' or *jobz* = 'O' and $m < n$. If *jobz* = 'S', the second dimension of *u* must be at least max(1, min(*m*,*n*)) . If *jobz* = 'A' or *jobz* = 'O' and $m < n$, *u* contains the *m*-by-*m* orthogonal/unitary matrix *U*. If *jobz* = 'S', *u* contains the first min(*m*,*n*) columns of *U* (the left singular vectors, stored columnwise). If *jobz* = 'O' and $m \ge n$, or *jobz* = 'N', *u* is not referenced. *vt*(*ldvt*,*); the second dimension of *vt* must be at least max(1, *n*). If *jobz* = 'A' or *jobz* = 'O' and $m \ge n$, *vt* contains the *n*-by-*n* orthogonal/unitary matrix V^T. If *jobz* = 'S', *vt* contains the first min(*m*,*n*) rows of V^T (the right singular vectors, stored rowwise). If *jobz* = 'O' and $m < n$, or *jobz* = 'N', *vt* is not referenced.

work(1) On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.
If *info* = *i*, then ?bdsdc did not converge, updating process failed.

Application Notes

For real flavors:
If jobz ='N', *lwork* $\geq 3*min(m,n) + max(max(m,n), 6*min(m,n))$;
If jobz ='O', *lwork* $\geq 3*(min(m,n))^2 + max(max(m,n), 5*(min(m,n))^2 + 4*min(m,n))$;
If jobz ='S' *or* 'A', *lwork* $\geq 3*(min(m,n))^2 + max(max(m,n), 4*(min(m,n))^2 + 4*min(m,n))$.

For complex flavors:
If jobz ='N', *lwork* $\geq 2*min(m,n) + max(m,n)$;
If jobz ='O', *lwork* $\geq 2*(min(m,n))^2 + max(m,n) + 2*min(m,n)$;
If jobz ='S' *or* 'A', *lwork* $\geq (min(m,n))^2 + max(m,n) + 2*min(m,n)$;

For good performance, *lwork* should generally be larger.
If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

?ggsvd

Computes the generalized singular value decomposition of a pair of general rectangular matrices.

Syntax

```
call sggsvd ( jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha,
              beta, u, ldu, v, ldv, q, ldq, work, iwork, info)
call dggsvd ( jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha,
              beta, u, ldu, v, ldv, q, ldq, work, iwork, info)
call cggsvd ( jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha,
              beta, u, ldu, v, ldv, q, ldq, work, rwork, iwork, info)
call zggsvd ( jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha,
              beta, u, ldu, v, ldv, q, ldq, work, rwork, iwork, info)
```

Description

This routine computes the generalized singular value decomposition (GSVD) of an *m*-by-*n* real/complex matrix *A* and *p*-by-*n* real/complex matrix *B*:

$$U^H A Q = D_1{*}(\,0 \quad R\,), \qquad V^H B Q = D_2{*}(\,0 \quad R\,),$$

where *U*, *V* and *Q* are orthogonal/unitary matrices.

Let $k+l$ = the effective numerical rank of the matrix $(A^H, B^H)^H$, then *R* is a $(k+l)$-by-$(k+l)$ nonsingular upper triangular matrix, D_1 and D_2 are *m*-by-$(k+l)$ and *p*-by-$(k+l)$ "diagonal" matrices and of the following structures, respectively:

If $m-k-l \geq 0$,

$$D_1 = \begin{array}{c} \\ k \\ l \\ m-k-l \end{array} \begin{array}{c} \begin{array}{cc} k & l \end{array} \\ \begin{pmatrix} I & 0 \\ 0 & C \\ 0 & 0 \end{pmatrix} \end{array}$$

$$D_2 = \begin{array}{c} \\ l \\ p-l \end{array} \begin{array}{c} \begin{array}{cc} k & l \end{array} \\ \begin{pmatrix} 0 & S \\ 0 & 0 \end{pmatrix} \end{array}$$

$$\begin{pmatrix} 0 & R \end{pmatrix} = \begin{array}{c} \\ k \\ l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & R_{11} & R_{12} \\ 0 & 0 & R_{22} \end{pmatrix} \end{array}$$

where

C = diag ($alpha(k+1),...,alpha(k+l)$)
S = diag ($beta(k+1),...,beta(k+l)$)
$C^2 + S^2 = I$

R is stored in $a(1:k+l, n-k-l+1:n)$ on exit.

If $m-k-l < 0$,

$$D_1 = \begin{array}{c} \\ k \\ m-k \end{array} \begin{array}{c} \begin{array}{ccc} k & m-k & k+l-m \end{array} \\ \begin{pmatrix} I & 0 & 0 \\ 0 & C & 0 \end{pmatrix} \end{array}$$

$$D_2 = \begin{array}{c} \\ m-k \\ k+l-m \\ p-l \end{array} \begin{array}{c} \begin{array}{ccc} k & m-k & k+l-m \end{array} \\ \begin{pmatrix} 0 & S & 0 \\ 0 & 0 & I \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$\begin{pmatrix} 0 & R \end{pmatrix} = \begin{array}{c} \\ k \\ m-k \\ k+l-m \end{array} \begin{array}{c} \begin{array}{cccc} n-k-l & k & m-k & k+l-m \end{array} \\ \begin{pmatrix} 0 & R_{11} & R_{12} & R_{13} \\ 0 & 0 & R_{22} & R_{23} \\ 0 & 0 & 0 & R_{33} \end{pmatrix} \end{array}$$

where

$C = \text{diag}(\, alpha(k+1),...,alpha(m))$,
$S = \text{diag}(\, beta(k+1),...,beta(m))$,
$C^2 + S^2 = I$

On exit, $\begin{pmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \end{pmatrix}$ is stored in $a(1{:}m,\ n-k-l+1{:}n)$ and R_{33} is stored in $b(m-k+1{:}l,\ n+m-k-l+1{:}n)$.

The routine computes *C*, *S*, *R*, and optionally the orthogonal/unitary transformation matrices *U*, *V* and *Q*.
In particular, if *B* is an *n*-by-*n* nonsingular matrix, then the GSVD of *A* and *B* implicitly gives the SVD of AB^{-1}:

$$AB^{-1} = U\,(D_1 D_2^{-1})\,V^H.$$

If $(A^H, B^H)^H$ has orthonormal columns, then the GSVD of *A* and *B* is also equal to the CS decomposition of *A* and *B*. Furthermore, the GSVD can be used to derive the solution of the eigenvalue problem:

$$A^H A\, x = \lambda\, B^H B\, x.$$

Input Parameters

jobu — `CHARACTER*1`. Must be `'U'` or `'N'`.
If *jobu* = `'U'`, orthogonal/unitary matrix *U* is computed.
If *jobu* = `'N'`, *U* is not computed.

jobv — `CHARACTER*1`. Must be `'V'` or `'N'`.
If *jobv* = `'V'`, orthogonal/unitary matrix *V* is computed.
If *jobv* = `'N'`, *V* is not computed.

jobq — `CHARACTER*1`. Must be `'Q'` or `'N'`.
If *jobq* = `'Q'`, orthogonal/unitary matrix *Q* is computed.
If *jobq* = `'N'`, *Q* is not computed.

m — `INTEGER`. The number of rows of the matrix *A* ($m \ge 0$).

n — `INTEGER`. The number of columns of the matrices *A* and *B* ($n \ge 0$).

p — `INTEGER`. The number of rows of the matrix *B* ($p \ge 0$).

a, b, work	`REAL` for `sggsvd` `DOUBLE PRECISION` for `dggsvd` `COMPLEX` for `cggsvd` `DOUBLE COMPLEX` for `zggsvd`. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the *p*-by-*n* matrix *B*. The second dimension of *b* must be at least max(1, *n*). *work*(*) is a workspace array. The dimension of *work* must be at least max(3*n*, *m*, *p*)+*n*.
lda	`INTEGER`. The first dimension of *a*; at least max(1, *m*).
ldb	`INTEGER`. The first dimension of *b*; at least max(1, *p*).
ldu	`INTEGER`. The first dimension of the array *u*. *ldu* ≥ max(1, *m*) if *jobu* = 'U'; *ldu* ≥ 1 otherwise.
ldv	`INTEGER`. The first dimension of the array *v*. *ldv* ≥ max(1, *p*) if *jobv* = 'V'; *ldv* ≥ 1 otherwise.
ldq	`INTEGER`. The first dimension of the array *q*. *ldq* ≥ max(1, *n*) if *jobq* = 'Q'; *ldq* ≥ 1 otherwise.
iwork	`INTEGER`. Workspace array, `DIMENSION` at least max(1, *n*).
rwork	`REAL` for `cggsvd` `DOUBLE PRECISION` for `zggsvd`. Workspace array, `DIMENSION` at least max(1, 2*n*). Used in complex flavors only.

Output Parameters

k, l	`INTEGER`. On exit, *k* and *l* specify the dimension of the subblocks. The sum *k*+*l* is equal to the effective numerical rank of $(A^H, B^H)^H$.
a	On exit, *a* contains the triangular matrix *R* or part of *R*.
b	On exit, *b* contains part of the triangular matrix *R* if *m*-*k*-*l* < 0.

alpha, beta

REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Arrays, DIMENSION at least max(1,*n*) each.
Contain the generalized singular value pairs of *A* and *B*:

alpha(1:*k*) = 1,
beta(1:*k*) = 0,

and if *m*-*k*-*l* ≥ 0,
alpha(*k*+1:*k*+*l*) = *C*,
beta(*k*+1:*k*+*l*) = *S*,

or if *m*-*k*-*l* < 0,
alpha(*k*+1:*m*)= *C*, *alpha*(*m*+1:*k*+*l*)= 0
beta(*k*+1:*m*) = *S*, *beta*(*m*+1:*k*+*l*) = 1

and
alpha(*k*+*l*+1:*n*) = 0
beta(*k*+*l*+1:*n*) = 0.

u, v, q

REAL for sggsvd
DOUBLE PRECISION for dggsvd
COMPLEX for cggsvd
DOUBLE COMPLEX for zggsvd.
Arrays:
u(*ldu*,*); the second dimension of *u* must be at least max(1, *m*).
If *jobu* ='U', *u* contains the *m*-by-*m* orthogonal/unitary matrix *U*.
If *jobu* ='N', *u* is not referenced.
v(*ldv*,*); the second dimension of *v* must be at least max(1, *p*).
If *jobv* ='V', *v* contains the *p*-by-*p* orthogonal/unitary matrix *V*.
If *jobv* ='N', *v* is not referenced.
q(*ldq*,*); the second dimension of *q* must be at least max(1, *n*).
If *jobq* ='Q', *q* contains the *n*-by-*n* orthogonal/unitary matrix *Q*.
If *jobq* ='N', *q* is not referenced.

iwork

On exit, *iwork* stores the sorting information.

info

INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = 1, the Jacobi-type procedure failed to converge. For further details, see subroutine ?tgsja.

Generalized Symmetric Definite Eigenproblems

This section describes LAPACK driver routines used for solving generalized symmetric definite eigenproblems. See also computational routines that can be called to solve these problems. Table 4-13 lists routines described in more detail below.

Table 4-13 Driver Routines for Solving Generalized Symmetric Definite Eigenproblems

Routine Name	Operation performed
?sygv /?hegv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem.
?sygvd/?hegvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem. If eigenvectors are desired, it uses a divide and conquer method.
?sygvx /?hegvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem.
?spgv/?hpgv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with matrices in packed storage.
?spgvd /?hpgvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with matrices in packed storage. If eigenvectors are desired, it uses a divide and conquer method.
?spgvx/?hpgvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with matrices in packed storage.
?sbgv /?hbgv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with banded matrices.
?sbgvd/?hbgvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.
?sbgvx/?hbgvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with banded matrices.

?sygv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

```
call ssygv ( itype, jobz, uplo, n, a, lda, b, ldb, w, work,
             lwork, info )
call dsygv ( itype, jobz, uplo, n, a, lda, b, ldb, w, work,
             lwork, info )
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \quad \text{or} \quad BAx = \lambda x .$$

Here A and B are assumed to be symmetric and B is also positive definite.

Input Parameters

itype — `INTEGER`. Must be 1 or 2 or 3.
Specifies the problem type to be solved:
if *itype* = 1, the problem type is $Ax = \lambda Bx$;
if *itype* = 2, the problem type is $ABx = \lambda x$;
if *itype* = 3, the problem type is $BAx = \lambda x$.

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then compute eigenvalues only.
If *jobz* = `'V'`, then compute eigenvalues and eigenvectors.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, arrays *a* and *b* store the upper triangles of A and B;
If *uplo* = `'L'`, arrays *a* and *b* store the lower triangles of A and B.

n — `INTEGER`. The order of the matrices A and B ($n \ge 0$).

a, b, work — `REAL` for `ssygv`
`DOUBLE PRECISION` for `dsygv`.
Arrays:

`a(lda,*)` contains the upper or lower triangle of the symmetric matrix A, as specified by `uplo`.
The second dimension of `a` must be at least max(1, `n`).

`b(ldb,*)` contains the upper or lower triangle of the symmetric positive definite matrix B, as specified by `uplo`.
The second dimension of `b` must be at least max(1, `n`).

`work(lwork)` is a workspace array.

lda — `INTEGER`. The first dimension of `a`; at least max(1, `n`).

ldb — `INTEGER`. The first dimension of `b`; at least max(1, `n`).

lwork — `INTEGER`. The dimension of the array `work`;
`lwork` ≥ max(1, 3`n`-1).
See *Application Notes* for the suggested value of `lwork`.

Output Parameters

a — On exit, if `jobz` ='V', then if `info` = 0, `a` contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:
if `itype` = 1 or 2, $Z^T B Z = I$;
if `itype` = 3, $Z^T B^{-1} Z = I$;

If `jobz` ='N', then on exit the upper triangle (if `uplo` = 'U') or the lower triangle (if `uplo` = 'L') of A, including the diagonal, is destroyed.

b — On exit, if `info` ≤ `n`, the part of `b` containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $B = U^T U$ or $B = L L^T$.

w — `REAL` for `ssygv`
`DOUBLE PRECISION` for `dsygv`.
Array, `DIMENSION` at least max(1, `n`).
If `info` = 0, contains the eigenvalues in ascending order.

work(1) — On exit, if `info` = 0, then `work(1)` returns the required minimal size of `lwork`.

info — `INTEGER`.
If `info` = 0, the execution is successful.
If `info` = `-i`, the `i`th argument had an illegal value.
If `info` > 0, `spotrf/dpotrf` and `ssyev/dsyev` returned an error code:

If $info = i \le n$, ssyev/dsyev failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;
If $info = n + i$, for $1 \le i \le n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Application Notes

For optimum performance use $lwork \ge (nb+2)*n$, where nb is the blocksize for ssytrd/dsytrd returned by ilaenv.
If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

?hegv

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem.

Syntax

```
call chegv ( itype, jobz, uplo, n, a, lda, b, ldb, w, work,
             lwork, rwork, info )
call zhegv ( itype, jobz, uplo, n, a, lda, b, ldb, w, work,
             lwork, rwork, info )
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \text{ or } BAx = \lambda x.$$

Here A and B are assumed to be Hermitian and B is also positive definite.

Input Parameters

itype	`INTEGER`. Must be 1 or 2 or 3. Specifies the problem type to be solved: if *itype* = 1, the problem type is $Ax = \lambda Bx$; if *itype* = 2, the problem type is $ABx = \lambda x$; if *itype* = 3, the problem type is $BAx = \lambda x$.
jobz	`CHARACTER*1`. Must be `'N'` or `'V'`. If *jobz* =`'N'`, then compute eigenvalues only. If *jobz* =`'V'`, then compute eigenvalues and eigenvectors.
uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. If *uplo* = `'U'`, arrays *a* and *b* store the upper triangles of A and B; If *uplo* = `'L'`, arrays *a* and *b* store the lower triangles of A and B.
n	`INTEGER`. The order of the matrices A and B ($n \ge 0$).
a, b, work	`COMPLEX` for `chegv` `DOUBLE COMPLEX` for `zhegv`. Arrays:

	`a(lda,*)` contains the upper or lower triangle of the Hermitian matrix A, as specified by *uplo*. The second dimension of *a* must be at least max(1, *n*).
	`b(ldb,*)` contains the upper or lower triangle of the Hermitian positive definite matrix B, as specified by *uplo*. The second dimension of *b* must be at least max(1, *n*).
	`work(lwork)` is a workspace array.
lda	`INTEGER`. The first dimension of *a*; at least max(1, *n*).
ldb	`INTEGER`. The first dimension of *b*; at least max(1, *n*).
lwork	`INTEGER`. The dimension of the array *work*; *lwork* ≥ max(1, 2*n*-1). See *Application Notes* for the suggested value of *lwork*.
rwork	`REAL` for `chegv` `DOUBLE PRECISION` for `zhegv`. Workspace array, `DIMENSION` at least max(1, 3*n*-2).

Output Parameters

a	On exit, if *jobz* ='V', then if *info* = 0, *a* contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if *itype* = 1 or 2, $Z^H B Z = I$; if *itype* = 3, $Z^H B^{-1} Z = I$; If *jobz* ='N', then on exit the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of A, including the diagonal, is destroyed.
b	On exit, if *info* ≤ *n*, the part of *b* containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $B = U^H U$ or $B = L L^H$.
w	`REAL` for `chegv` `DOUBLE PRECISION` for `zhegv`. Array, `DIMENSION` at least max(1, *n*). If *info* = 0, contains the eigenvalues in ascending order.
work(1)	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = *-i*, the *i*th argument had an illegal value. If *info* > 0, `cpotrf/zpotrf` and `cheev/zheev` returned an error code:

If *info* = *i* ≤ *n*, `cheev/zheev` failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;
If *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Application Notes

For optimum performance use *lwork* ≥ (*nb*+1)**n*, where *nb* is the blocksize for `chetrd/zhetrd` returned by `ilaenv`.
If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

?sygvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

```
call ssygvd ( itype, jobz, uplo, n, a, lda, b, ldb, w, work,
              lwork, iwork, liwork, info )
call dsygvd ( itype, jobz, uplo, n, a, lda, b, ldb, w, work,
              lwork, iwork, liwork, info )
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \text{ or } BAx = \lambda x.$$

Here A and B are assumed to be symmetric and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

itype — `INTEGER`. Must be 1 or 2 or 3.
Specifies the problem type to be solved:
if *itype* = 1, the problem type is $Ax = \lambda Bx$;
if *itype* = 2, the problem type is $ABx = \lambda x$;
if *itype* = 3, the problem type is $BAx = \lambda x$.

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* =`'N'`, then compute eigenvalues only.
If *jobz* =`'V'`, then compute eigenvalues and eigenvectors.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, arrays *a* and *b* store the upper triangles of A and B;
If *uplo* = `'L'`, arrays *a* and *b* store the lower triangles of A and B.

n — `INTEGER`. The order of the matrices A and B ($n \geq 0$).

a, b, work — `REAL` for `ssygvd`
`DOUBLE PRECISION` for `dsygvd`.
Arrays:
a(*lda*,*) contains the upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*.
The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the upper or lower triangle of the symmetric positive definite matrix *B*, as specified by *uplo*.
The second dimension of *b* must be at least max(1, *n*).

work(*lwork*) is a workspace array.

lda — `INTEGER`. The first dimension of *a*; at least max(1, *n*).

ldb — `INTEGER`. The first dimension of *b*; at least max(1, *n*).

lwork — `INTEGER`. The dimension of the array *work*.

Constraints:
If $n \le 1$, *lwork* ≥ 1;
If *jobz* ='N' and $n>1$, *lwork* $\ge 2n+1$;
If *jobz* ='V' and $n>1$, *lwork* $\ge 2n^2+6n+1$.

iwork — `INTEGER`.
Workspace array, `DIMENSION` (*liwork*).

liwork — `INTEGER`. The dimension of the array *iwork*.
Constraints:
If $n \le 1$, *liwork* ≥ 1;
If *jobz* ='N' and $n>1$, *liwork* ≥ 1;
If *jobz* ='V' and $n>1$, *liwork* $\ge 5n+3$.

Output Parameters

a — On exit, if *jobz* ='V', then if *info* = 0, *a* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:
if *itype* = 1 or 2, $Z^T B Z = I$;
if *itype* = 3, $Z^T B^{-1} Z = I$;

If *jobz* ='N', then on exit the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of *A*, including the diagonal, is destroyed.

b — On exit, if *info* ≤ *n*, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^T U$ or $B = L L^T$.

w	`REAL` for `ssygvd` `DOUBLE PRECISION` for `dsygvd`. Array, `DIMENSION` at least max(1, *n*). If *info* = 0, contains the eigenvalues in ascending order.
work(1)	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
iwork(1)	On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = *-i*, the *i*th argument had an illegal value. If *info* > 0, `spotrf/dpotrf` and `ssyev/dsyev` returned an error code: If *info* = $i \le n$, `ssyev/dsyev` failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero; If *info* = $n + i$, for $1 \le i \le n$, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

?hegvd

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

```
call chegvd ( itype, jobz, uplo, n, a, lda, b, ldb, w, work,
              lwork, rwork, lrwork, iwork, liwork, info )
call zhegvd ( itype, jobz, uplo, n, a, lda, b, ldb, w, work,
              lwork, rwork, lrwork, iwork, liwork, info )
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \text{ or } BAx = \lambda x.$$

Here A and B are assumed to be Hermitian and B is also positive definite. If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

itype — `INTEGER`. Must be 1 or 2 or 3.
Specifies the problem type to be solved:
if *itype* = 1, the problem type is $Ax = \lambda Bx$;
if *itype* = 2, the problem type is $ABx = \lambda x$;
if *itype* = 3, the problem type is $BAx = \lambda x$.

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then compute eigenvalues only.
If *jobz* = `'V'`, then compute eigenvalues and eigenvectors.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, arrays *a* and *b* store the upper triangles of A and B;
If *uplo* = `'L'`, arrays *a* and *b* store the lower triangles of A and B.

n — `INTEGER`. The order of the matrices A and B ($n \ge 0$).

a, b, work — COMPLEX for chegvd
DOUBLE COMPLEX for zhegvd.
Arrays:
a(*lda*,*) contains the upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*.
The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the upper or lower triangle of the Hermitian positive definite matrix *B*, as specified by *uplo*.
The second dimension of *b* must be at least max(1, *n*).

work(*lwork*) is a workspace array.

lda — INTEGER. The first dimension of *a*; at least max(1, *n*).

ldb — INTEGER. The first dimension of *b*; at least max(1, *n*).

lwork — INTEGER. The dimension of the array *work*.

Constraints:
If $n \le 1$, $lwork \ge 1$;
If *jobz* ='N' and $n>1$, $lwork \ge n+1$;
If *jobz* ='V' and $n>1$, $lwork \ge n^2+2n$.

rwork — REAL for chegvd
DOUBLE PRECISION for zhegvd.
Workspace array, DIMENSION (*lrwork*).

lrwork — INTEGER. The dimension of the array *rwork*.
Constraints:
If $n \le 1$, $lrwork \ge 1$;
If *jobz* ='N' and $n>1$, $lrwork \ge n$;
If *jobz* ='V' and $n>1$, $lrwork \ge 2n^2+5n+1$.

iwork — INTEGER.
Workspace array, DIMENSION (*liwork*). .

liwork — INTEGER. The dimension of the array *iwork*.
Constraints:
If $n \le 1$, $liwork \ge 1$;
If *jobz* ='N' and $n>1$, $liwork \ge 1$;
If *jobz* ='V' and $n>1$, $liwork \ge 5n+3$.

Output Parameters

a
On exit, if *jobz* = 'V', then if *info* = 0, *a* contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:
if *itype* = 1 or 2, $Z^H B Z = I$;
if *itype* = 3, $Z^H B^{-1} Z = I$;

If *jobz* = 'N', then on exit the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of A, including the diagonal, is destroyed.

b
On exit, if *info* ≤ *n*, the part of *b* containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $B = U^H U$ or $B = L L^H$.

w
REAL for chegvd
DOUBLE PRECISION for zhegvd.
Array, DIMENSION at least max(1, *n*).
If *info* = 0, contains the eigenvalues in ascending order.

work(1)
On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

rwork(1)
On exit, if *info* = 0, then *rwork(1)* returns the required minimal size of *lrwork*.

iwork(1)
On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.

info
INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th argument had an illegal value.
If *info* > 0, cpotrf/zpotrf and cheev/zheev returned an error code:

If *info* = *i* ≤ *n*, cheev/zheev failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;
If *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then the leading minor of order *i* of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

?sygvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

```
call ssygvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il,
            iu, abstol, m, w, z, ldz, work, lwork, iwork, ifail, info)
call dsygvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il,
            iu, abstol, m, w, z, ldz, work, lwork, iwork, ifail, info)
```

Description

This routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \text{ or } BAx = \lambda x.$$

Here A and B are assumed to be symmetric and B is also positive definite.
Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

itype — `INTEGER`. Must be 1 or 2 or 3.
Specifies the problem type to be solved:
if *itype* = 1, the problem type is $Ax = \lambda Bx$;
if *itype* = 2, the problem type is $ABx = \lambda x$;
if *itype* = 3, the problem type is $BAx = \lambda x$.

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then compute eigenvalues only.
If *jobz* = `'V'`, then compute eigenvalues and eigenvectors.

range — `CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`.
If *range* = `'A'`, the routine computes all eigenvalues.
If *range* = `'V'`, the routine computes eigenvalues λ_i in the half-open interval: $vl < \lambda_i \le vu$.
If *range* = `'I'`, the routine computes eigenvalues with indices *il* to *iu*.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, arrays *a* and *b* store the upper triangles of *A* and *B*;
If *uplo* = `'L'`, arrays *a* and *b* store the lower triangles of *A* and *B*.

n — `INTEGER`. The order of the matrices *A* and *B* ($n \ge 0$).

a, b, work — `REAL` for `ssygvx`
`DOUBLE PRECISION` for `dsygvx`.
Arrays:
a(*lda*,*) contains the upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*.
The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) contains the upper or lower triangle of the symmetric positive definite matrix *B*, as specified by *uplo*.
The second dimension of *b* must be at least max(1, *n*).

work(*lwork*) is a workspace array.

lda — `INTEGER`. The first dimension of *a*; at least max(1, *n*).

ldb — `INTEGER`. The first dimension of *b*; at least max(1, *n*).

vl, vu — `REAL` for `ssygvx`
`DOUBLE PRECISION` for `dsygvx`.
If *range* =`'V'`, the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: $vl < vu$.

If *range* =`'A'` or `'I'`, *vl* and *vu* are not referenced.

il, iu — `INTEGER`.
If *range* =`'I'`, the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: $1 \le il \le iu \le n$, if $n > 0$; il=1 and iu=0 if $n = 0$.

If *range* =`'A'` or `'V'`, *il* and *iu* are not referenced.

abstol — `REAL` for `ssygvx`
`DOUBLE PRECISION` for `dsygvx`.
The absolute error tolerance for the eigenvalues.
See *Application Notes* for more information.

ldz — `INTEGER`. The leading dimension of the output array *z*. Constraints:
$ldz \ge 1$; if *jobz* =`'V'`, $ldz \ge \max(1, n)$.

lwork — INTEGER. The dimension of the array *work*;
lwork ≥ max(1, 8*n*).
See *Application Notes* for the suggested value of *lwork*.

iwork — INTEGER.
Workspace array, DIMENSION at least max(1, 5*n*).

Output Parameters

a — On exit, the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of *A*, including the diagonal, is overwritten.

b — On exit, if *info* ≤ *n*, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^T U$ or $B = L L^T$.

m — INTEGER. The total number of eigenvalues found,
0 ≤ *m* ≤ *n*. If *range* = 'A', *m* = *n*, and if *range* = 'I',
m = *iu*-*il*+1.

w, z — REAL for ssygvx
DOUBLE PRECISION for dsygvx.
Arrays:
w(*), DIMENSION at least max(1, *n*) .
The first *m* elements of *w* contain the selected eigenvalues in ascending order.

z(ldz,)* . The second dimension of *z* must be at least max(1, *m*).
If *jobz* ='V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized as follows:
if *itype* = 1 or 2, $Z^T B Z = I$;
if *itype* = 3, $Z^T B^{-1} Z = I$;

If *jobz* ='N', then *z* is not referenced.
If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z* ; if *range* ='V', the exact value of *m* is not known in advance and an upper bound must be used.

work(1) — On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

ifail — INTEGER.
Array, DIMENSION at least max(1, *n*).
If *jobz* ='V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.
If *jobz* ='N', then *ifail* is not referenced.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th argument had an illegal value.
If *info* > 0, spotrf/dpotrf and ssyevx/dsyevx returned an error code:

If *info* = *i* $\le n$, ssyevx/dsyevx failed to converge, and *i* eigenvectors failed to converge. Their indices are stored in the array *ifail*;
If *info* = *n* + *i*, for $1 \le i \le n$, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to
abstol + ε * max(|a|,|b|) , where ε is the machine precision. If *abstol* is less than or equal to zero, then $\varepsilon * \|T\|_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form.
Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

For optimum performance use *lwork* ≥ (*nb*+3)**n*, where *nb* is the blocksize for ssytrd/dsytrd returned by ilaenv.
If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

?hegvx

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem.

Syntax

```
call chegvx ( itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu,
              il, iu, abstol, m, w, z, ldz, work, lwork, rwork,
               iwork, ifail, info)
call zhegvx ( itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu,
              il, iu, abstol, m, w, z, ldz, work, lwork, rwork,
               iwork, ifail, info)
```

Description

This routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \text{ or } BAx = \lambda x .$$

Here A and B are assumed to be Hermitian and B is also positive definite.
Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

itype — `INTEGER`. Must be 1 or 2 or 3.
Specifies the problem type to be solved:
if *itype* = 1, the problem type is $Ax = \lambda Bx$;
if *itype* = 2, the problem type is $ABx = \lambda x$;
if *itype* = 3, the problem type is $BAx = \lambda x$.

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then compute eigenvalues only.
If *jobz* = `'V'`, then compute eigenvalues and eigenvectors.

range — `CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`.
If *range* = `'A'`, the routine computes all eigenvalues.
If *range* = `'V'`, the routine computes eigenvalues λ_i in the half-open interval: $vl < \lambda_i \le vu$.
If *range* = `'I'`, the routine computes eigenvalues with indices *il* to *iu*.

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. If *uplo* = `'U'`, arrays *a* and *b* store the upper triangles of *A* and *B*; If *uplo* = `'L'`, arrays *a* and *b* store the lower triangles of *A* and *B*.
n	`INTEGER`. The order of the matrices *A* and *B* ($n \geq 0$).
a, b, work	`COMPLEX` for `chegvx` `DOUBLE COMPLEX` for `zhegvx`. Arrays: *a*(*lda*,*) contains the upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the upper or lower triangle of the Hermitian positive definite matrix *B*, as specified by *uplo*. The second dimension of *b* must be at least max(1, *n*). *work*(*lwork*) is a workspace array.
lda	`INTEGER`. The first dimension of *a*; at least max(1, *n*).
ldb	`INTEGER`. The first dimension of *b*; at least max(1, *n*).
vl, vu	`REAL` for `chegvx` `DOUBLE PRECISION` for `zhegvx`. If *range* =`'V'`, the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If *range* =`'A'` or `'I'`, *vl* and *vu* are not referenced.
il, iu	`INTEGER`. If *range* =`'I'`, the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; il=1 and iu=0 if $n = 0$. If *range* =`'A'` or `'V'`, *il* and *iu* are not referenced.
abstol	`REAL` for `chegvx` `DOUBLE PRECISION` for `zhegvx`. The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.
ldz	`INTEGER`. The leading dimension of the output array *z*. Constraints: $ldz \geq 1$; if *jobz* =`'V'`, $ldz \geq \max(1, n)$.

lwork INTEGER. The dimension of the array *work*;
$lwork \geq \max(1, 2n-1)$.
See *Application Notes* for the suggested value of *lwork*.

rwork REAL for chegvx
DOUBLE PRECISION for zhegvx.
Workspace array, DIMENSION at least $\max(1, 7n)$.

iwork INTEGER.
Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

a On exit, the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of *A*, including the diagonal, is overwritten.

b On exit, if $info \leq n$, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^H U$ or $B = L L^H$.

m INTEGER. The total number of eigenvalues found,
$0 \leq m \leq n$. If *range* ='A', $m = n$, and if *range* ='I',
$m = iu-il+1$.

w REAL for chegvx
DOUBLE PRECISION for zhegvx.
Array, DIMENSION at least $\max(1, n)$.
The first *m* elements of *w* contain the selected eigenvalues in ascending order.

z COMPLEX for chegvx
DOUBLE COMPLEX for zhegvx.
Array z(*ldz*,*) . The second dimension of *z* must be at least $\max(1, m)$.
If *jobz* ='V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized as follows:
if *itype* = 1 or 2, $Z^H B Z = I$;
if *itype* = 3, $Z^H B^{-1} Z = I$;

If *jobz* ='N', then *z* is not referenced.
If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z* ; if *range* ='V', the exact value of *m* is not known in advance and an upper bound must be used.

work(1) — On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

ifail — INTEGER.
Array, DIMENSION at least max(1, *n*).
If *jobz* ='V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.
If *jobz* ='N', then *ifail* is not referenced.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th argument had an illegal value.
If *info* > 0, cpotrf/zpotrf and cheevx/zheevx returned an error code:

If *info* = *i* $\le n$, cheevx/zheevx failed to converge, and *i* eigenvectors failed to converge. Their indices are stored in the array *ifail*;
If *info* = *n* + *i*, for $1 \le i \le n$, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to
$abstol + \varepsilon * \max(|a|,|b|)$, where ε is the machine precision. If *abstol* is less than or equal to zero, then $\varepsilon * \|T\|_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form.
Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

For optimum performance use *lwork* ≥ (*nb*+1)**n*, where *nb* is the blocksize for chetrd/zhetrd returned by ilaenv.
If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

?spgv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage.

Syntax

```
call sspgv ( itype, jobz, uplo, n, ap, bp, w, z, ldz, work, info )
call dspgv ( itype, jobz, uplo, n, ap, bp, w, z, ldz, work, info )
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \text{ or } BAx = \lambda x.$$

Here A and B are assumed to be symmetric, stored in packed format, and B is also positive definite.

Input Parameters

itype — `INTEGER`. Must be 1 or 2 or 3.
Specifies the problem type to be solved:
if *itype* = 1, the problem type is $Ax = \lambda Bx$;
if *itype* = 2, the problem type is $ABx = \lambda x$;
if *itype* = 3, the problem type is $BAx = \lambda x$.

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then compute eigenvalues only.
If *jobz* = `'V'`, then compute eigenvalues and eigenvectors.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, arrays *ap* and *bp* store the upper triangles of A and B;
If *uplo* = `'L'`, arrays *ap* and *bp* store the lower triangles of A and B.

n — `INTEGER`. The order of the matrices A and B ($n \geq 0$).

ap, bp, work — `REAL` for `sspgv`
`DOUBLE PRECISION` for `dspgv`.
Arrays:
ap(*) contains the packed upper or lower triangle of the symmetric matrix A, as specified by *uplo*. The dimension of *ap* must be at least max(1, $n*(n+1)/2$).

bp(*) contains the packed upper or lower triangle of the symmetric matrix *B*, as specified by *uplo*. The dimension of *bp* must be at least max(1, *n**(*n*+1)/2).

work(*) is a workspace array, `DIMENSION` at least max(1, 3*n*).

ldz — `INTEGER`. The leading dimension of the output array *z*; *ldz* ≥ 1. If *jobz* ='V', *ldz* ≥ max(1, *n*) .

Output Parameters

ap — On exit, the contents of *ap* are overwritten.

bp — On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^T U$ or $B = L L^T$, in the same storage format as *B*.

w, z — `REAL` for `sspgv`
`DOUBLE PRECISION` for `dspgv`.
Arrays:
w(*), `DIMENSION` at least max(1, *n*) .
If *info* = 0, contains the eigenvalues in ascending order.

z(ldz,)* . The second dimension of *z* must be at least max(1, *n*).
If *jobz* ='V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:
if *itype* = 1 or 2, $Z^T B Z = I$;
if *itype* = 3, $Z^T B^{-1} Z = I$;

If *jobz* ='N', then *z* is not referenced.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th argument had an illegal value.
If *info* > 0, `spptrf/dpptrf` and `sspev/dspev` returned an error code:

If *info* = *i* ≤ *n*, `sspev/dspev` failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;
If *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

?hpgv

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with matrices in packed storage.

Syntax

```
call chpgv ( itype, jobz, uplo, n, ap, bp, w, z, ldz, work, rwork,
             info )
call zhpgv ( itype, jobz, uplo, n, ap, bp, w, z, ldz, work, rwork,
             info )
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$Ax = \lambda Bx$, $ABx = \lambda x$, or $BAx = \lambda x$.

Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite.

Input Parameters

itype	`INTEGER`. Must be 1 or 2 or 3. Specifies the problem type to be solved: if *itype* = 1, the problem type is $Ax = \lambda Bx$; if *itype* = 2, the problem type is $ABx = \lambda x$; if *itype* = 3, the problem type is $BAx = \lambda x$.
jobz	`CHARACTER*1`. Must be `'N'` or `'V'`. If *jobz* =`'N'`, then compute eigenvalues only. If *jobz* =`'V'`, then compute eigenvalues and eigenvectors.
uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. If *uplo* = `'U'`, arrays *ap* and *bp* store the upper triangles of A and B; If *uplo* = `'L'`, arrays *ap* and *bp* store the lower triangles of A and B.
n	`INTEGER`. The order of the matrices A and B ($n \geq 0$).
ap, bp, work	`COMPLEX` for `chpgv` `DOUBLE COMPLEX` for `zhpgv`. Arrays:

ap(*) contains the packed upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*. The dimension of *ap* must be at least max(1, *n**(*n*+1)/2).

bp(*) contains the packed upper or lower triangle of the Hermitian matrix *B*, as specified by *uplo*. The dimension of *bp* must be at least max(1, *n**(*n*+1)/2).

work(*) is a workspace array, DIMENSION at least max(1, 2*n*-1).

ldz — INTEGER. The leading dimension of the output array *z*; *ldz* ≥ 1. If *jobz* ='V', *ldz* ≥ max(1, *n*) .

rwork — REAL for chpgv
DOUBLE PRECISION for zhpgv.
Workspace array, DIMENSION at least max(1, 3*n*-2).

Output Parameters

ap — On exit, the contents of *ap* are overwritten.

bp — On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^H U$ or $B = L L^H$, in the same storage format as *B*.

w — REAL for chpgv
DOUBLE PRECISION for zhpgv.
Array, DIMENSION at least max(1, *n*) .
If *info* = 0, contains the eigenvalues in ascending order.

z — COMPLEX for chpgv
DOUBLE COMPLEX for zhpgv.
Array *z(ldz,*)*. The second dimension of *z* must be at least max(1, *n*).
If *jobz* ='V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:
if *itype* = 1 or 2, $Z^H B Z = I$;
if *itype* = 3, $Z^H B^{-1} Z = I$;

If *jobz* ='N', then *z* is not referenced.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th argument had an illegal value.
If *info* > 0, cpptrf/zpptrf and chpev/zhpev returned an error code:

If *info* = $i \le n$, chpev/zhpev failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;
If *info* = $n + i$, for $1 \le i \le n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

?spgvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

```
call sspgvd ( itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork,
              iwork, liwork, info )
call dspgvd ( itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork,
              iwork, liwork, info )
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \text{ or } BAx = \lambda x .$$

Here A and B are assumed to be symmetric, stored in packed format, and B is also positive definite. If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

itype — `INTEGER`. Must be 1 or 2 or 3.
Specifies the problem type to be solved:
if *itype* = 1, the problem type is $Ax = \lambda Bx$;
if *itype* = 2, the problem type is $ABx = \lambda x$;
if *itype* = 3, the problem type is $BAx = \lambda x$.

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then compute eigenvalues only.
If *jobz* = `'V'`, then compute eigenvalues and eigenvectors.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, arrays *ap* and *bp* store the upper triangles of A and B;
If *uplo* = `'L'`, arrays *ap* and *bp* store the lower triangles of A and B.

n — `INTEGER`. The order of the matrices A and B ($n \geq 0$).

ap, bp, work — REAL for sspgvd
DOUBLE PRECISION for dspgvd.
Arrays:
ap(*) contains the packed upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*. The dimension of *ap* must be at least max(1, $n*(n+1)/2$).

bp(*) contains the packed upper or lower triangle of the symmetric matrix *B*, as specified by *uplo*. The dimension of *bp* must be at least max(1, $n*(n+1)/2$).

work(lwork) is a workspace array.

ldz — INTEGER. The leading dimension of the output array *z*; $ldz \geq 1$. If *jobz* ='V', $ldz \geq \max(1, n)$.

lwork — INTEGER. The dimension of the array *work*.

Constraints:
If $n \leq 1$, $lwork \geq 1$;
If *jobz* ='N' and $n>1$, $lwork \geq 2n$;
If *jobz* ='V' and $n>1$, $lwork \geq 2n^2+6n+1$.

iwork — INTEGER.
Workspace array, DIMENSION (*liwork*). .

liwork — INTEGER. The dimension of the array *iwork*.
Constraints:
If $n \leq 1$, $liwork \geq 1$;
If *jobz* ='N' and $n>1$, $liwork \geq 1$;
If *jobz* ='V' and $n>1$, $liwork \geq 5n+3$.

Output Parameters

ap — On exit, the contents of *ap* are overwritten.

bp — On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^TU$ or $B = L L^T$, in the same storage format as *B*.

w, z — REAL for sspgv
DOUBLE PRECISION for dspgv.
Arrays:
w(*), DIMENSION at least max(1, *n*).
If *info* = 0, contains the eigenvalues in ascending order.

z(ldz,)*. The second dimension of *z* must be at least max(1, *n*).
If *jobz* ='V', then if *info* = 0, *z* contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:
if *itype* = 1 or 2, $Z^T B Z = I$;
if *itype* = 3, $Z^T B^{-1} Z = I$;

If *jobz* ='N', then *z* is not referenced.

work(1) On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

iwork(1) On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th argument had an illegal value.
If *info* > 0, spptrf/dpptrf and sspevd/dspevd returned an error code:

If *info* = $i \le n$, sspevd/dspevd failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;
If *info* = $n + i$, for $1 \le i \le n$, then the leading minor of order *i* of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

?hpgvd

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with matrices in packed storage. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

```
call chpgvd ( itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork,
              rwork, lrwork, iwork, liwork, info )
call zhpgvd ( itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork,
              rwork, lrwork, iwork, liwork, info )
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \quad \text{or} \quad BAx = \lambda x .$$

Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite. If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

itype — `INTEGER`. Must be 1 or 2 or 3.
Specifies the problem type to be solved:
if *itype* = 1, the problem type is $Ax = \lambda Bx$;
if *itype* = 2, the problem type is $ABx = \lambda x$;
if *itype* = 3, the problem type is $BAx = \lambda x$.

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then compute eigenvalues only.
If *jobz* = `'V'`, then compute eigenvalues and eigenvectors.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, arrays *ap* and *bp* store the upper triangles of A and B;
If *uplo* = `'L'`, arrays *ap* and *bp* store the lower triangles of A and B.

n — `INTEGER`. The order of the matrices A and B ($n \ge 0$).

ap, bp, work — `COMPLEX` for `chpgvd`
`DOUBLE COMPLEX` for `zhpgvd`.
Arrays:
ap(*) contains the packed upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*. The dimension of *ap* must be at least max(1, *n**(*n*+1)/2).

bp(*) contains the packed upper or lower triangle of the Hermitian matrix *B*, as specified by *uplo*. The dimension of *bp* must be at least max(1, *n**(*n*+1)/2).

work(lwork) is a workspace array.

ldz — `INTEGER`. The leading dimension of the output array *z*; $ldz \geq 1$. If *jobz* ='V', $ldz \geq \max(1, n)$.

lwork — `INTEGER`. The dimension of the array *work*.

Constraints:
If $n \leq 1$, $lwork \geq 1$;
If *jobz* ='N' and $n>1$, $lwork \geq n$;
If *jobz* ='V' and $n>1$, $lwork \geq 2n$.

rwork — `REAL` for `chpgvd`
`DOUBLE PRECISION` for `zhpgvd`.
Workspace array, `DIMENSION` (*lrwork*).

lrwork — `INTEGER`. The dimension of the array *rwork*.
Constraints:
If $n \leq 1$, $lrwork \geq 1$;
If *jobz* ='N' and $n>1$, $lrwork \geq n$;
If *jobz* ='V' and $n>1$, $lrwork \geq 2n^2+5n+1$.

iwork — `INTEGER`.
Workspace array, `DIMENSION` (*liwork*). .

liwork — `INTEGER`. The dimension of the array *iwork*.
Constraints:
If $n \leq 1$, $liwork \geq 1$;
If *jobz* ='N' and $n>1$, $liwork \geq 1$;
If *jobz* ='V' and $n>1$, $liwork \geq 5n+3$.

Output Parameters

ap — On exit, the contents of *ap* are overwritten.

bp	On exit, contains the triangular factor U or L from the Cholesky factorization $B = U^H U$ or $B = L L^H$, in the same storage format as B.
w	`REAL` for `chpgvd` `DOUBLE PRECISION` for `zhpgvd`. Array, `DIMENSION` at least max(1, *n*) . If *info* = 0, contains the eigenvalues in ascending order.
z	`COMPLEX` for `chpgvd` `DOUBLE COMPLEX` for `zhpgvd`. Array `z(ldz,*)` . The second dimension of *z* must be at least max(1, *n*). If *jobz* ='V', then if *info* = 0, *z* contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if *itype* = 1 or 2, $Z^H B Z = I$; if *itype* = 3, $Z^H B^{-1} Z = I$; If *jobz* ='N', then *z* is not referenced.
work(1)	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
rwork(1)	On exit, if *info* = 0, then *rwork(1)* returns the required minimal size of *lrwork*.
iwork(1)	On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = *-i*, the *i*th argument had an illegal value. If *info* > 0, `cpptrf/zpptrf` and `chpevd/zhpevd` returned an error code: If *info* = $i \le n$, `chpevd/zhpevd` failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero; If *info* = $n + i$, for $1 \le i \le n$, then the leading minor of order *i* of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

?spgvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage.

Syntax

```
call sspgvx ( itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu,
              abstol, m, w, z, ldz, work, iwork, ifail, info )
call dspgvx ( itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu,
              abstol, m, w, z, ldz, work, iwork, ifail, info )
```

Description

This routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$Ax = \lambda Bx,\quad ABx = \lambda x,\ \text{or}\ BAx = \lambda x .$$

Here A and B are assumed to be symmetric, stored in packed format, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

itype — `INTEGER`. Must be 1 or 2 or 3.
Specifies the problem type to be solved:
if *itype* = 1, the problem type is $Ax = \lambda Bx$;
if *itype* = 2, the problem type is $ABx = \lambda x$;
if *itype* = 3, the problem type is $BAx = \lambda x$.

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then compute eigenvalues only.
If *jobz* = `'V'`, then compute eigenvalues and eigenvectors.

range — `CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`.
If *range* = `'A'`, the routine computes all eigenvalues.
If *range* = `'V'`, the routine computes eigenvalues λ_i in the half-open interval: $vl < \lambda_i \le vu$.
If *range* = `'I'`, the routine computes eigenvalues with indices *il* to *iu*.

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. If *uplo* = `'U'`, arrays *ap* and *bp* store the upper triangles of *A* and *B*; If *uplo* = `'L'`, arrays *ap* and *bp* store the lower triangles of *A* and *B*.
n	`INTEGER`. The order of the matrices *A* and *B* ($n \geq 0$).
ap, bp, work	`REAL` for `sspgvx` `DOUBLE PRECISION` for `dspgvx`. Arrays: *ap(*)* contains the packed upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*. The dimension of *ap* must be at least max(1, $n*(n+1)/2$). *bp(*)* contains the packed upper or lower triangle of the symmetric matrix *B*, as specified by *uplo*. The dimension of *bp* must be at least max(1, $n*(n+1)/2$). *work(*)* is a workspace array, `DIMENSION` at least max(1, 8*n*).
vl, vu	`REAL` for `sspgvx` `DOUBLE PRECISION` for `dspgvx`. If *range* =`'V'`, the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: *vl*< *vu*. If *range* =`'A'` or `'I'`, *vl* and *vu* are not referenced.
il, iu	`INTEGER`. If *range* =`'I'`, the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; *il*=1 and *iu*=0 if $n = 0$. If *range* =`'A'` or `'V'`, *il* and *iu* are not referenced.
abstol	`REAL` for `sspgvx` `DOUBLE PRECISION` for `dspgvx`. The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.
ldz	`INTEGER`. The leading dimension of the output array *z*. Constraints: $ldz \geq 1$; if *jobz* =`'V'`, $ldz \geq \max(1, n)$.
iwork	`INTEGER`. Workspace array, `DIMENSION` at least max(1, 5*n*).

Output Parameters

ap — On exit, the contents of *ap* are overwritten.

bp — On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^T U$ or $B = L L^T$, in the same storage format as *B*.

m — INTEGER. The total number of eigenvalues found, $0 \le m \le n$. If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu*-*il*+1.

w, z — REAL for sspgvx
DOUBLE PRECISION for dspgvx.
Arrays:
w(*), DIMENSION at least max(1, *n*) .
If *info* = 0, contains the eigenvalues in ascending order.

z(*ldz*,*) . The second dimension of *z* must be at least max(1, *n*).
If *jobz* ='V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized as follows:
if *itype* = 1 or 2, $Z^T B Z = I$;
if *itype* = 3, $Z^T B^{-1} Z = I$;

If *jobz* ='N', then *z* is not referenced.
If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z* ; if *range* ='V', the exact value of *m* is not known in advance and an upper bound must be used.

ifail — INTEGER.
Array, DIMENSION at least max(1, *n*).
If *jobz* ='V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.
If *jobz* ='N', then *ifail* is not referenced.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th argument had an illegal value.
If *info* > 0, spptrf/dpptrf and sspevx/dspevx returned an error code:

If *info* = $i \le n$, `sspevx/dspevx` failed to converge, and i eigenvectors failed to converge. Their indices are stored in the array *ifail*;
If *info* = $n + i$, for $1 \le i \le n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to
abstol + ε * max(|a|,|b|) , where ε is the machine precision. If *abstol* is less than or equal to zero, then $\varepsilon * \|T\|_1$ will be used in its place, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form.
Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*`?lamch`('S'), not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*`?lamch`('S').

?hpgvx

Computes selected eigenvalues and, optionally, eigenvectors of a generalized Hermitian definite eigenproblem with matrices in packed storage.

Syntax

```
call chpgvx ( itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu,
              abstol, m, w, z, ldz, work, rwork, iwork, ifail, info )
call zhpgvx ( itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu,
              abstol, m, w, z, ldz, work, rwork, iwork, ifail, info )
```

Description

This routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$Ax = \lambda Bx, \quad ABx = \lambda x, \text{ or } BAx = \lambda x$.

Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

itype — `INTEGER`. Must be 1 or 2 or 3.
Specifies the problem type to be solved:
if *itype* = 1, the problem type is $Ax = \lambda Bx$;
if *itype* = 2, the problem type is $ABx = \lambda x$;
if *itype* = 3, the problem type is $BAx = \lambda x$.

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then compute eigenvalues only.
If *jobz* = `'V'`, then compute eigenvalues and eigenvectors.

range — `CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`.
If *range* = `'A'`, the routine computes all eigenvalues.
If *range* = `'V'`, the routine computes eigenvalues λ_i in the half-open interval: $vl < \lambda_i \le vu$.
If *range* = `'I'`, the routine computes eigenvalues with indices *il* to *iu*.

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. If *uplo* = `'U'`, arrays *ap* and *bp* store the upper triangles of *A* and *B*; If *uplo* = `'L'`, arrays *ap* and *bp* store the lower triangles of *A* and *B*.
n	`INTEGER`. The order of the matrices *A* and *B* ($n \geq 0$).
ap, bp, work	`COMPLEX` for `chpgvx` `DOUBLE COMPLEX` for `zhpgvx`. Arrays: *ap(*)* contains the packed upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*. The dimension of *ap* must be at least max(1, $n*(n+1)/2$). *bp(*)* contains the packed upper or lower triangle of the Hermitian matrix *B*, as specified by *uplo*. The dimension of *bp* must be at least max(1, $n*(n+1)/2$). *work(*)* is a workspace array, `DIMENSION` at least max(1, 2*n*).
vl, vu	`REAL` for `chpgvx` `DOUBLE PRECISION` for `zhpgvx`. If *range* =`'V'`, the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: *vl*< *vu*. If *range* =`'A'` or `'I'`, *vl* and *vu* are not referenced.
il, iu	`INTEGER`. If *range* =`'I'`, the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; *il*=1 and *iu*=0 if $n = 0$. If *range* =`'A'` or `'V'`, *il* and *iu* are not referenced.
abstol	`REAL` for `chpgvx` `DOUBLE PRECISION` for `zhpgvx`. The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.
ldz	`INTEGER`. The leading dimension of the output array *z*; $ldz \geq 1$. If *jobz* =`'V'`, $ldz \geq \max(1, n)$.
rwork	`REAL` for `chpgvx` `DOUBLE PRECISION` for `zhpgvx`. Workspace array, `DIMENSION` at least max(1, 7*n*).

iwork INTEGER.
Workspace array, DIMENSION at least max(1, 5n).

Output Parameters

ap On exit, the contents of *ap* are overwritten.

bp On exit, contains the triangular factor U or L from the Cholesky factorization $B = U^H U$ or $B = L L^H$, in the same storage format as B.

m INTEGER. The total number of eigenvalues found,
$0 \le m \le n$. If *range* ='A', $m = n$, and if *range* ='I',
m = *iu*-*il*+1.

w REAL for chpgvx
DOUBLE PRECISION for zhpgvx.
Array, DIMENSION at least max(1, n) .
If *info* = 0, contains the eigenvalues in ascending order.

z COMPLEX for chpgvx
DOUBLE COMPLEX for zhpgvx.
Array z(*ldz*,*) . The second dimension of *z* must be at least max(1, n).
If *jobz* ='V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized as follows:
if *itype* = 1 or 2, $Z^H B Z = I$;
if *itype* = 3, $Z^H B^{-1} Z = I$;

If *jobz* ='N', then *z* is not referenced.

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z* ; if *range* ='V', the exact value of *m* is not known in advance and an upper bound must be used.

ifail INTEGER.
Array, DIMENSION at least max(1, n).
If *jobz* ='V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.
If *jobz* ='N', then *ifail* is not referenced.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th argument had an illegal value.
If *info* > 0, cpptrf/zpptrf and chpevx/zhpevx returned an error code:

If *info* = $i \le n$, chpevx/zhpevx failed to converge, and *i* eigenvectors failed to converge. Their indices are stored in the array *ifail*;
If *info* = $n + i$, for $1 \le i \le n$, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to
$abstol + \varepsilon * \max(|a|,|b|)$, where ε is the machine precision. If *abstol* is less than or equal to zero, then $\varepsilon * \|T\|_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form.
Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?sbgv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices.

Syntax

```
call ssbgv ( jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz,
             work, info )
call dsbgv ( jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz,
             work, info )
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form
$Ax = \lambda Bx$. Here A and B are assumed to be symmetric and banded, and B is also positive definite.

Input Parameters

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* =`'N'`, then compute eigenvalues only.
If *jobz* =`'V'`, then compute eigenvalues and eigenvectors.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, arrays *ab* and *bb* store the upper triangles of A and B;
If *uplo* = `'L'`, arrays *ab* and *bb* store the lower triangles of A and B.

n — `INTEGER`. The order of the matrices A and B ($n \ge 0$).

ka — `INTEGER`. The number of super- or sub-diagonals in A ($ka \ge 0$).

kb — `INTEGER`. The number of super- or sub-diagonals in B ($kb \ge 0$).

ab, bb, work — `REAL` for `ssbgv`
`DOUBLE PRECISION` for `dsbgv`
Arrays:
ab (*ldab*,*) is an array containing either upper or lower triangular part of the symmetric matrix A (as specified by *uplo*) in band storage format.
The second dimension of the array *ab* must be at least max(1, *n*).

	bb (*ldbb*, *) is an array containing either upper or lower triangular part of the symmetric matrix B (as specified by *uplo*) in band storage format. The second dimension of the array *bb* must be at least max(1, *n*). *work(*)* is a workspace array, `DIMENSION` at least max(1, 3*n*)
ldab	`INTEGER`. The first dimension of the array *ab*; must be at least *ka*+1.
ldbb	`INTEGER`. The first dimension of the array *bb*; must be at least *kb*+1.
ldz	`INTEGER`. The leading dimension of the output array *z*; *ldz* ≥ 1. If *jobz* ='V', *ldz* ≥ max(1, *n*) .

Output Parameters

ab	On exit, the contents of *ab* are overwritten.
bb	On exit, contains the factor S from the split Cholesky factorization $B = S^TS$, as returned by `spbstf/dpbstf`.
w, z	`REAL` for `ssbgv` `DOUBLE PRECISION` for `dsbgv` Arrays: *w*(*), `DIMENSION` at least max(1, *n*) . If *info* = 0, contains the eigenvalues in ascending order. *z(ldz, *)* . The second dimension of *z* must be at least max(1, *n*). If *jobz* ='V', then if *info* = 0, *z* contains the matrix Z of eigenvectors , with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^TB\,Z = I$. If *jobz* ='N', then *z* is not referenced.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th argument had an illegal value. If *info* > 0, and if *i* ≤ *n*, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero; if *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then `spbstf/dpbstf` returned *info* = *i* and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

?hbgv

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with banded matrices.

Syntax

```
call chbgv ( jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz,
             work, rwork, info )
call zhbgv ( jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz,
             work, rwork, info )
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form $Ax = \lambda Bx$. Here A and B are assumed to be Hermitian and banded, and B is also positive definite.

Input Parameters

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* =`'N'`, then compute eigenvalues only.
If *jobz* =`'V'`, then compute eigenvalues and eigenvectors.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, arrays *ab* and *bb* store the upper triangles of A and B;
If *uplo* = `'L'`, arrays *ab* and *bb* store the lower triangles of A and B.

n — `INTEGER`. The order of the matrices A and B ($n \geq 0$).

ka — `INTEGER`. The number of super- or sub-diagonals in A ($ka \geq 0$).

kb — `INTEGER`. The number of super- or sub-diagonals in B ($kb \geq 0$).

ab, bb, work — `COMPLEX` for `chbgv`
`DOUBLE COMPLEX` for `zhbgv`
Arrays:
ab (*ldab*, *) is an array containing either upper or lower triangular part of the Hermitian matrix A (as specified by *uplo*) in band storage format.
The second dimension of the array *ab* must be at least max(1, *n*).

bb (*ldbb*, *) is an array containing either upper or lower triangular part of the Hermitian matrix *B* (as specified by *uplo*) in band storage format.
The second dimension of the array *bb* must be at least max(1, *n*).
work(*) is a workspace array, DIMENSION at least max(1, *n*) .

ldab INTEGER. The first dimension of the array *ab*; must be at least *ka*+1.

ldbb INTEGER. The first dimension of the array *bb*; must be at least *kb*+1.

ldz INTEGER. The leading dimension of the output array *z*; *ldz* ≥ 1. If *jobz* ='V', *ldz* ≥ max(1, *n*) .

rwork REAL for chbgv
DOUBLE PRECISION for zhbgv.
Workspace array, DIMENSION at least max(1, 3*n*).

Output Parameters

ab On exit, the contents of *ab* are overwritten.

bb On exit, contains the factor *S* from the split Cholesky factorization $B = S^H S$, as returned by cpbstf/zpbstf.

w REAL for chbgv
DOUBLE PRECISION for zhbgv.
Array, DIMENSION at least max(1, *n*) .
If *info* = 0, contains the eigenvalues in ascending order.

z COMPLEX for chbgv
DOUBLE COMPLEX for zhbgv
Array z(*ldz*, *) . The second dimension of *z* must be at least max(1, *n*).
If *jobz* ='V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors , with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^H B Z = I$.
If *jobz* ='N', then *z* is not referenced.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th argument had an illegal value.
If *info* > 0, and

if *i* ≤ *n*, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;
if *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then cpbstf/zpbstf returned *info* = *i* and *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

?sbgvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

```
call ssbgvd ( jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz,
              work, lwork, iwork, liwork, info )
call dsbgvd ( jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz,
              work, lwork, iwork, liwork, info )
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form
$Ax = \lambda Bx$. Here A and B are assumed to be symmetric and banded, and B is also positive definite. If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then compute eigenvalues only.
If *jobz* = `'V'`, then compute eigenvalues and eigenvectors.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, arrays *ab* and *bb* store the upper triangles of A and B;
If *uplo* = `'L'`, arrays *ab* and *bb* store the lower triangles of A and B.

n — `INTEGER`. The order of the matrices A and B ($n \geq 0$).

ka — `INTEGER`. The number of super- or sub-diagonals in A ($ka \geq 0$).

kb — `INTEGER`. The number of super- or sub-diagonals in B ($kb \geq 0$).

ab, bb, work — `REAL` for `ssbgvd`
`DOUBLE PRECISION` for `dsbgvd`
Arrays:

	ab (*ldab*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format. The second dimension of the array *ab* must be at least max(1, *n*).
	bb (*ldbb*,*) is an array containing either upper or lower triangular part of the symmetric matrix *B* (as specified by *uplo*) in band storage format. The second dimension of the array *bb* must be at least max(1, *n*).
	work(*lwork*) is a workspace array.
ldab	INTEGER. The first dimension of the array *ab*; must be at least *ka*+1.
ldbb	INTEGER. The first dimension of the array *bb*; must be at least *kb*+1.
ldz	INTEGER. The leading dimension of the output array *z*; $ldz \ge 1$. If *jobz* ='V', $ldz \ge \max(1, n)$.
lwork	INTEGER. The dimension of the array *work*. Constraints: If $n \le 1$, $lwork \ge 1$; If *jobz* ='N' and $n>1$, $lwork \ge 3n$; If *jobz* ='V' and $n>1$, $lwork \ge 2n^2+5n+1$.
iwork	INTEGER. Workspace array, DIMENSION (*liwork*). .
liwork	INTEGER. The dimension of the array *iwork*. Constraints: If $n \le 1$, $liwork \ge 1$; If *jobz* ='N' and $n>1$, $liwork \ge 1$; If *jobz* ='V' and $n>1$, $liwork \ge 5n+3$.

Output Parameters

ab	On exit, the contents of *ab* are overwritten.
bb	On exit, contains the factor *S* from the split Cholesky factorization $B = S^TS$, as returned by spbstf/dpbstf.
w, z	REAL for ssbgvd DOUBLE PRECISION for dsbgvd Arrays: *w*(*), DIMENSION at least max(1, *n*) . If *info* = 0, contains the eigenvalues in ascending order.

`z(ldz,*)`. The second dimension of `z` must be at least max(1, `n`).
If `jobz` ='V', then if `info` = 0, `z` contains the matrix Z of eigenvectors , with the `i`-th column of `z` holding the eigenvector associated with `w(i)`. The eigenvectors are normalized so that $Z^T B Z = I$.
If `jobz` ='N', then `z` is not referenced.

work(1) — On exit, if `info` = 0, then `work(1)` returns the required minimal size of `lwork`.

iwork(1) — On exit, if `info` = 0, then `iwork(1)` returns the required minimal size of `liwork`.

info — INTEGER.
If `info` = 0, the execution is successful.
If `info` = `-i`, the `i`th argument had an illegal value.
If `info` > 0, and

if `i` ≤ `n`, the algorithm failed to converge, and `i` off-diagonal elements of an intermediate tridiagonal did not converge to zero;
if `info` = `n` + `i`, for 1 ≤ `i` ≤ `n`, then spbstf/dpbstf returned `info` = `i` and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

?hbgvd

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

```
call chbgvd ( jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz,
              work, lwork, rwork, lrwork, iwork, liwork, info )
call zhbgvd ( jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz,
              work, lwork, rwork, lrwork, iwork, liwork, info )
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form $Ax = \lambda Bx$. Here A and B are assumed to be Hermitian and banded, and B is also positive definite. If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

jobz	`CHARACTER*1`. Must be `'N'` or `'V'`. If *jobz* =`'N'`, then compute eigenvalues only. If *jobz* =`'V'`, then compute eigenvalues and eigenvectors.
uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. If *uplo* = `'U'`, arrays *ab* and *bb* store the upper triangles of A and B; If *uplo* = `'L'`, arrays *ab* and *bb* store the lower triangles of A and B.
n	`INTEGER`. The order of the matrices A and B ($n \ge 0$).
ka	`INTEGER`. The number of super- or sub-diagonals in A ($ka \ge 0$).
kb	`INTEGER`. The number of super- or sub-diagonals in B ($kb \ge 0$).
ab, bb, work	`COMPLEX` for `chbgvd` `DOUBLE COMPLEX` for `zhbgvd` Arrays:

ab (*ldab*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format.
The second dimension of the array *ab* must be at least max(1, *n*).

bb (*ldbb*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *B* (as specified by *uplo*) in band storage format.
The second dimension of the array *bb* must be at least max(1, *n*).

work(*lwork*) is a workspace array.

ldab INTEGER. The first dimension of the array *ab*; must be at least *ka*+1.

ldbb INTEGER. The first dimension of the array *bb*; must be at least *kb*+1.

ldz INTEGER. The leading dimension of the output array *z*; $ldz \geq 1$. If *jobz* ='V', $ldz \geq \max(1, n)$.

lwork INTEGER. The dimension of the array *work*.

Constraints:
If $n \leq 1$, $lwork \geq 1$;
If *jobz* ='N' and $n>1$, $lwork \geq n$;
If *jobz* ='V' and $n>1$, $lwork \geq 2n^2$.

rwork REAL for chbgvd
DOUBLE PRECISION for zhbgvd.
Workspace array, DIMENSION (*lrwork*).

lrwork INTEGER. The dimension of the array *rwork*.

Constraints:
If $n \leq 1$, $lrwork \geq 1$;
If *jobz* ='N' and $n>1$, $lrwork \geq n$;
If *jobz* ='V' and $n>1$, $lrwork \geq 2n^2+5n+1$.

iwork INTEGER.
Workspace array, DIMENSION (*liwork*).

liwork INTEGER. The dimension of the array *iwork*.
Constraints:
If $n \leq 1$, $liwork \geq 1$;
If *jobz* ='N' and $n>1$, $liwork \geq 1$;
If *jobz* ='V' and $n>1$, $liwork \geq 5n+3$.

Output Parameters

ab On exit, the contents of *ab* are overwritten.

bb	On exit, contains the factor S from the split Cholesky factorization $B = S^H S$, as returned by `cpbstf/zpbstf`.
w	`REAL` for `chbgvd` `DOUBLE PRECISION` for `zhbgvd`. Array, `DIMENSION` at least max(1, *n*) . If *info* = 0, contains the eigenvalues in ascending order.
z	`COMPLEX` for `chbgvd` `DOUBLE COMPLEX` for `zhbgvd` Array `z(ldz,*)` . The second dimension of *z* must be at least max(1, *n*). If *jobz* ='V', then if *info* = 0, *z* contains the matrix Z of eigenvectors , with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^H B Z = I$. If *jobz* ='N', then *z* is not referenced.
work(1)	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
rwork(1)	On exit, if *info* = 0, then *rwork(1)* returns the required minimal size of *lrwork*.
iwork(1)	On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th argument had an illegal value. If *info* > 0, and if $i \le n$, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero; if $info = n + i$, for $1 \le i \le n$, then `cpbstf/zpbstf` returned *info* = *i* and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

?sbgvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices.

Syntax

```
call ssbgvx ( jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q,
              ldq, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork,
              ifail, info )
call dsbgvx ( jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q,
              ldq, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork,
              ifail, info )
```

Description

This routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form
$Ax = \lambda Bx$. Here A and B are assumed to be symmetric and banded, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then compute eigenvalues only.
If *jobz* = `'V'`, then compute eigenvalues and eigenvectors.

range — `CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`.
If *range* = `'A'`, the routine computes all eigenvalues.
If *range* = `'V'`, the routine computes eigenvalues λ_i in the half-open interval: $vl < \lambda_i \le vu$.
If *range* = `'I'`, the routine computes eigenvalues with indices *il* to *iu*.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.

If *uplo* = `'U'`, arrays *ab* and *bb* store the upper triangles of A and B;
If *uplo* = `'L'`, arrays *ab* and *bb* store the lower triangles of A and B.

n — `INTEGER`. The order of the matrices A and B ($n \ge 0$).

ka	INTEGER. The number of super- or sub-diagonals in *A* ($ka \ge 0$).
kb	INTEGER. The number of super- or sub-diagonals in *B* ($kb \ge 0$).
ab,bb,work	REAL for ssbgvx DOUBLE PRECISION for dsbgvx Arrays: *ab* (*ldab*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format. The second dimension of the array *ab* must be at least max(1, *n*). *bb* (*ldbb*,*) is an array containing either upper or lower triangular part of the symmetric matrix *B* (as specified by *uplo*) in band storage format. The second dimension of the array *bb* must be at least max(1, *n*). *work(*)* is a workspace array, DIMENSION at least max(1, 7*n*).
ldab	INTEGER. The first dimension of the array *ab*; must be at least *ka*+1.
ldbb	INTEGER. The first dimension of the array *bb*; must be at least *kb*+1.
vl, vu	REAL for ssbgvx DOUBLE PRECISION for dsbgvx. If *range* ='V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: *vl*< *vu*. If *range* ='A' or 'I', *vl* and *vu* are not referenced.
il, iu	INTEGER. If *range* ='I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \le il \le iu \le n$, if $n > 0$; *il*=1 and *iu*=0 if $n = 0$. If *range* ='A' or 'V', *il* and *iu* are not referenced.
abstol	REAL for ssbgvx DOUBLE PRECISION for dsbgvx. The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.
ldz	INTEGER. The leading dimension of the output array *z*; $ldz \ge 1$. If *jobz* ='V', $ldz \ge \max(1, n)$.

ldq INTEGER. The leading dimension of the output array *q*; $ldq \geq 1$. If *jobz* ='V', $ldq \geq \max(1, n)$.

iwork INTEGER.
Workspace array, DIMENSION at least max(1, 5*n*).

Output Parameters

ab On exit, the contents of *ab* are overwritten.

bb On exit, contains the factor *S* from the split Cholesky factorization $B = S^T S$, as returned by spbstf/dpbstf.

m INTEGER. The total number of eigenvalues found,
$0 \leq m \leq n$. If *range* ='A', *m* = *n*, and if *range* ='I',
m = *iu*-*il*+1.

w, z, q REAL for ssbgvx
DOUBLE PRECISION for dsbgvx
Arrays:
w(*), DIMENSION at least max(1, *n*).
If *info* = 0, contains the eigenvalues in ascending order.

z(*ldz*,*). The second dimension of *z* must be at least max(1, *n*).
If *jobz* ='V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^T B Z = I$.
If *jobz* ='N', then *z* is not referenced.
q(*ldq*,*). The second dimension of *q* must be at least max(1, *n*).
If *jobz* ='V', then *q* contains the *n*-by-*n* matrix used in the reduction of $Ax = \lambda Bx$ to standard form, that is,
$Cx = \lambda x$ and consequently *C* to tridiagonal form.
If *jobz* ='N', then *q* is not referenced.

ifail INTEGER.
Array, DIMENSION at least max(1, *n*).
If *jobz* ='V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.
If *jobz* ='N', then *ifail* is not referenced.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th argument had an illegal value.
If *info* > 0, and

if $i \le n$, the algorithm failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;
if $info = n + i$, for $1 \le i \le n$, then `spbstf/dpbstf` returned $info = i$ and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to
$abstol + \varepsilon * \max(|a|,|b|)$, where ε is the machine precision. If *abstol* is less than or equal to zero, then $\varepsilon * \|T\|_1$ will be used in its place, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form.
Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?hbgvx

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with banded matrices.

Syntax

```
call chbgvx ( jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q,
              ldq, vl, vu, il, iu, abstol, m, w, z, ldz, work, rwork,
              iwork, ifail, info )
call zhbgvx ( jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q,
              ldq, vl, vu, il, iu, abstol, m, w, z, ldz, work, rwork,
              iwork, ifail, info )
```

Description

This routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form $Ax = \lambda Bx$. Here A and B are assumed to be Hermitian and banded, and B is also positive definite.
Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then compute eigenvalues only.
If *jobz* = `'V'`, then compute eigenvalues and eigenvectors.

range — `CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`.
If *range* = `'A'`, the routine computes all eigenvalues.
If *range* = `'V'`, the routine computes eigenvalues λ_i in the half-open interval: $vl < \lambda_i \le vu$.
If *range* = `'I'`, the routine computes eigenvalues with indices *il* to *iu*.

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.

If *uplo* = `'U'`, arrays *ab* and *bb* store the upper triangles of A and B;
If *uplo* = `'L'`, arrays *ab* and *bb* store the lower triangles of A and B.

n — `INTEGER`. The order of the matrices A and B ($n \ge 0$).

ka	INTEGER. The number of super- or sub-diagonals in *A* (*ka* ≥ 0).
kb	INTEGER. The number of super- or sub-diagonals in *B* (*kb* ≥ 0).
ab,bb,work	COMPLEX for chbgvx DOUBLE COMPLEX for zhbgvx Arrays: *ab* (*ldab*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format. The second dimension of the array *ab* must be at least max(1, *n*). *bb* (*ldbb*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *B* (as specified by *uplo*) in band storage format. The second dimension of the array *bb* must be at least max(1, *n*). *work(*)* is a workspace array, DIMENSION at least max(1, *n*).
ldab	INTEGER. The first dimension of the array *ab*; must be at least *ka*+1.
ldbb	INTEGER. The first dimension of the array *bb*; must be at least *kb*+1.
vl, vu	REAL for chbgvx DOUBLE PRECISION for zhbgvx. If *range* ='V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: *vl*< *vu*. If *range* ='A' or 'I', *vl* and *vu* are not referenced.
il, iu	INTEGER. If *range* ='I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: 1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0. If *range* ='A' or 'V', *il* and *iu* are not referenced.
abstol	REAL for chbgvx DOUBLE PRECISION for zhbgvx. The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.
ldz	INTEGER. The leading dimension of the output array *z*; *ldz* ≥ 1. If *jobz* ='V', *ldz* ≥ max(1, *n*) .

ldq — `INTEGER`. The leading dimension of the output array *q*; *ldq* ≥ 1. If *jobz* = `'V'`, *ldq* ≥ max(1, *n*) .

rwork — `REAL` for `chbgvx`
`DOUBLE PRECISION` for `zhbgvx`.
Workspace array, `DIMENSION` at least max(1, 7*n*).

iwork — `INTEGER`.
Workspace array, `DIMENSION` at least max(1, 5*n*).

Output Parameters

ab — On exit, the contents of *ab* are overwritten.

bb — On exit, contains the factor *S* from the split Cholesky factorization $B = S^H S$, as returned by `cpbstf/zpbstf`.

m — `INTEGER`. The total number of eigenvalues found, 0 ≤ *m* ≤ *n*. If *range* = `'A'`, *m* = *n*, and if *range* = `'I'`, *m* = *iu*-*il*+1.

w — `REAL` for `chbgvx`
`DOUBLE PRECISION` for `zhbgvx`.
Array *w*(*), `DIMENSION` at least max(1, *n*) .
If *info* = 0, contains the eigenvalues in ascending order.

z, q — `COMPLEX` for `chbgvx`
`DOUBLE COMPLEX` for `zhbgvx`
Arrays:
`z(`*ldz*, *`)` . The second dimension of *z* must be at least max(1, *n*).
If *jobz* = `'V'`, then if *info* = 0, *z* contains the matrix *Z* of eigenvectors , with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^H B Z = I$.
If *jobz* = `'N'`, then *z* is not referenced.
`q(`*ldq*, *`)` . The second dimension of *q* must be at least max(1, *n*).
If *jobz* = `'V'`, then *q* contains the *n*-by-*n* matrix used in the reduction of $Ax = \lambda Bx$ to standard form, that is,
$Cx = \lambda x$ and consequently *C* to tridiagonal form.
If *jobz* = `'N'`, then *q* is not referenced.

ifail — `INTEGER`.
Array, `DIMENSION` at least max(1, *n*).
If *jobz* = `'V'`, then if *info* = 0, the first *m* elements of *ifail* are zero; if

$info > 0$, the *ifail* contains the indices of the eigenvectors that failed to converge.
If *jobz* ='N', then *ifail* is not referenced.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th argument had an illegal value.
If *info* > 0, and

if $i \le n$, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;
if *info* = $n + i$, for $1 \le i \le n$, then cpbstf/zpbstf returned *info* = *i* and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to
abstol + ε * max(|a|,|b|) , where ε is the machine precision. If *abstol* is less than or equal to zero, then $\varepsilon * \|T\|_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form.
Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero. If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

Generalized Nonsymmetric Eigenproblems

This section describes LAPACK driver routines used for solving generalized nonsymmetric eigenproblems. See also computational routines that can be called to solve these problems. Table 4-14 lists routines described in more detail below.

Table 4-14 Driver Routines for Solving Generalized Nonsymmetric Eigenproblems

Routine Name	Operation performed
?gges	Computes the generalized eigenvalues, Schur form, and the left and/or right Schur vectors for a pair of nonsymmetric matrices.
?ggesx	Computes the generalized eigenvalues, Schur form, and, optionally, the left and/or right matrices of Schur vectors .
?ggev	Computes the generalized eigenvalues, and the left and/or right generalized eigenvectors for a pair of nonsymmetric matrices.
?ggevx	Computes the generalized eigenvalues, and, optionally, the left and/or right generalized eigenvectors.

?gges

Computes the generalized eigenvalues, Schur form, and the left and/or right Schur vectors for a pair of nonsymmetric matrices.

Syntax

```
call sgges ( jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim,
             alphar, alphai, beta, vsl, ldvsl, vsr, ldvsr, work,
              lwork, bwork, info )
call dgges ( jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim,
             alphar, alphai, beta, vsl, ldvsl, vsr, ldvsr, work,
              lwork, bwork, info )
call cgges ( jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim,
             alpha, beta, vsl, ldvsl, vsr, ldvsr, work, lwork, rwork,
             bwork, info )
call zgges ( jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim,
             alpha, beta, vsl, ldvsl, vsr, ldvsr, work, lwork, rwork,
             bwork, info )
```

Description

This routine computes for a pair of *n*-by-*n* real/complex nonsymmetric matrices (*A,B*), the generalized eigenvalues, the generalized real/complex Schur form (*S,T*), optionally, the left and/or right matrices of Schur vectors (*vsl* and *vsr*). This gives the generalized Schur factorization

$$(A,B) = (vsl*S*vsr^H, vsl*T*vsr^H)$$

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix *S* and the upper triangular matrix *T*. The leading columns of *vsl* and *vsr* then form an orthonormal/unitary basis for the corresponding left and right eigenspaces (deflating subspaces).
(If only the generalized eigenvalues are needed, use the driver `?ggev` instead, which is faster.)
A generalized eigenvalue for a pair of matrices (*A,B*) is a scalar *w* or a ratio *alpha* / *beta* = *w*, such that *A* - *w***B* is singular. It is usually represented as the pair (*alpha, beta*), as there is a reasonable interpretation for *beta*=0 or for both being zero.
A pair of matrices (*S,T*) is in generalized real Schur form if *T* is upper triangular with non-negative diagonal and *S* is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of *S* will be “standardized" by making the corresponding elements of *T* have the form:

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$$

and the pair of corresponding 2-by-2 blocks in *S* and *T* will have a complex conjugate pair of generalized eigenvalues.
A pair of matrices (*S,T*) is in generalized complex Schur form if *S* and *T* are upper triangular and, in addition, the diagonal of *T* are non-negative real numbers.

Input Parameters

jobvsl — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobvsl* =`'N'`, then the left Schur vectors are not computed.
If *jobvsl* =`'V'`, then the left Schur vectors are computed.

jobvsr — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobvsr* =`'N'`, then the right Schur vectors are not computed.
If *jobvsr* =`'V'`, then the right Schur vectors are computed.

sort — `CHARACTER*1`. Must be `'N'` or `'S'`.
Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form.

If `sort` ='N', then eigenvalues are not ordered.
If `sort` ='S', eigenvalues are ordered (see `selctg`).

`selctg` — `LOGICAL FUNCTION` of three `REAL` arguments for real flavors.
`LOGICAL FUNCTION` of two `COMPLEX` arguments for complex flavors.

`selctg` must be declared `EXTERNAL` in the calling subroutine.
If `sort` ='S', `selctg` is used to select eigenvalues to sort to the top left of the Schur form.
If `sort` ='N', `selctg` is not referenced.

For real flavors:
An eigenvalue (`alphar`(j) + `alphai`(j))/`beta`(j) is selected if `selctg`(`alphar`(j), `alphai`(j), `beta`(j)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.
Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy `selctg`(`alphar`(j), `alphai`(j), `beta`(j)) = `.TRUE.` after ordering. In this case `info` is set to n+2 .

For complex flavors:
An eigenvalue `alpha`(j) / `beta`(j) is selected if `selctg`(`alpha`(j), `beta`(j)) is true.
Note that a selected complex eigenvalue may no longer satisfy `selctg`(`alpha`(j), `beta`(j)) = `.TRUE.` after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case `info` is set to n+2 (see `info` below).

`n` — `INTEGER`. The order of the matrices A, B, `vsl`, and `vsr` ($n \ge 0$).

`a, b, work` — `REAL` for `sgges`
`DOUBLE PRECISION` for `dgges`
`COMPLEX` for `cgges`
`DOUBLE COMPLEX` for `zgges`.
Arrays:
`a(lda,*)` is an array containing the n-by-n matrix A (first of the pair of matrices).
The second dimension of `a` must be at least max(1, n).

	b(*ldb*,*) is an array containing the *n*-by-*n* matrix *B* (second of the pair of matrices). The second dimension of *b* must be at least max(1, *n*). *work*(*lwork*) is a workspace array.
lda	INTEGER. The first dimension of the array *a*. Must be at least max(1, *n*) .
ldb	INTEGER. The first dimension of the array *b*. Must be at least max(1, *n*) .
ldvsl, *ldvsr*	INTEGER. The first dimensions of the output matrices *vsl* and *vsr*, respectively. Constraints: *ldvsl* ≥ 1. If *jobvsl* ='V', *ldvsl* ≥ max(1, *n*) . *ldvsr* ≥ 1. If *jobvsr* ='V', *ldvsr* ≥ max(1, *n*) .
lwork	INTEGER. The dimension of the array *work*. *lwork* ≥ max(1, 8*n*+16) for real flavors; *lwork* ≥ max(1, 2*n*) for complex flavors. For good performance, *lwork* must generally be larger.
rwork	REAL for cgges DOUBLE PRECISION for zgges Workspace array, DIMENSION at least max(1, 8*n*). This array is used in complex flavors only.
bwork	LOGICAL. Workspace array, DIMENSION at least max(1, *n*). Not referenced if *sort* ='N'.

Output Parameters

a	On exit, this array has been overwritten by its generalized Schur form *S* .
b	On exit, this array has been overwritten by its generalized Schur form *T* .
sdim	INTEGER. If *sort* ='N', *sdim*= 0. If *sort* ='S', *sdim* is equal to the number of eigenvalues (after sorting) for which *selctg* is true. Note that for real flavors complex conjugate pairs for which *selctg* is true for either eigenvalue count as 2.

alphar,alphai REAL for sgges;
DOUBLE PRECISION for dgges.
Arrays, DIMENSION at least max(1,*n*) each. Contain values that form generalized eigenvalues in real flavors.
See *beta*.

alpha COMPLEX for cgges;
DOUBLE COMPLEX for zgges.
Array, DIMENSION at least max(1,*n*). Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta REAL for sgges
DOUBLE PRECISION for dgges
COMPLEX for cgges
DOUBLE COMPLEX for zgges.
Array, DIMENSION at least max(1,*n*).
For real flavors:
On exit, (*alphar*(j) + *alphai*(j)*i)/*beta*(j), j=1,...,*n*, will be the generalized eigenvalues.
alphar(j) + *alphai*(j)*i and *beta*(j), j=1,...,*n* are the diagonals of the complex Schur form (*S,T*) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (*A,B*) were further reduced to triangular form using complex unitary transformations. If *alphai*(j) is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with *alphai*(j+1) negative.
For complex flavors:
On exit, *alpha*(j)/*beta*(j), j=1,...,*n*, will be the generalized eigenvalues.
alpha(j), j=1,...,*n*, and *beta*(j), j=1,...,*n*, are the diagonals of the complex Schur form (*S,T*) output by cgges/zgges. The *beta*(j) will be non-negative real.

See also *Application Notes* below.

vsl, vsr REAL for sgges
DOUBLE PRECISION for dgges
COMPLEX for cgges
DOUBLE COMPLEX for zgges.
Arrays:
vsl(*ldvsl*,*), the second dimension of *vsl* must be at least max(1, *n*).
If *jobvsl* ='V', this array will contain the left Schur vectors.
If *jobvsl* ='N', *vsl* is not referenced.

vsr(*ldvsr*, *), the second dimension of *vsr* must be at least max(1, *n*).
If *jobvsr* ='V', this array will contain the right Schur vectors.
If *jobvsr* ='N', *vsr* is not referenced.

work(1) — On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, and
$i \le n$:

the *QZ* iteration failed. (*A,B*) is not in Schur form, but *alphar*(j), *alphai*(j) (for real flavors), or *alpha*(j) (for complex flavors), and *beta*(j), j=*info*+1,...,*n* should be correct.

$i > n$: errors that usually indicate LAPACK problems:

i = *n*+1: other than *QZ* iteration failed in ?hgeqz;

i = *n*+2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the generalized Schur form no longer satisfy *selctg* = .TRUE.. This could also be caused due to scaling;

i = *n*+3: reordering failed in ?tgsen.

Application Notes

If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The quotients *alphar*(j)/*beta*(j) and *alphai*(j)/*beta*(j) may easily over- or underflow, and *beta*(j) may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* will be always less than and usually comparable with norm(*A*) in magnitude, and *beta* always less than and usually comparable with norm(*B*).

?ggesx

Computes the generalized eigenvalues, Schur form, and, optionally, the left and/or right matrices of Schur vectors .

Syntax

```
call sggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb,
        sdim, alphar, alphai, beta, vsl, ldvsl, vsr, ldvsr,
         rconde, rcondv, work, lwork, iwork, liwork, bwork, info )
call dggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb,
        sdim, alphar, alphai, beta, vsl, ldvsl, vsr, ldvsr,
         rconde, rcondv, work, lwork, iwork, liwork, bwork, info )
call cggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb,
        sdim, alpha, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv,
        work, lwork, rwork, iwork, liwork, bwork, info )
call zggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb,
        sdim, alpha, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv,
        work, lwork, rwork, iwork, liwork, bwork, info )
```

Description

This routine computes for a pair of *n*-by-*n* real/complex nonsymmetric matrices (*A,B*), the generalized eigenvalues, the generalized real/complex Schur form (*S,T*), optionally, the left and/or right matrices of Schur vectors (*vsl* and *vsr*). This gives the generalized Schur factorization

$$(A,B) = (vsl*S*vsr^H, vsl*T*vsr^H)$$

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix *S* and the upper triangular matrix *T*; computes a reciprocal condition number for the average of the selected eigenvalues (*rconde*); and computes a reciprocal condition number for the right and left deflating subspaces corresponding to the selected eigenvalues (*rcondv*). The leading columns of *vsl* and *vsr* then form an orthonormal/unitary basis for the corresponding left and right eigenspaces (deflating subspaces).

A generalized eigenvalue for a pair of matrices (*A,B*) is a scalar *w* or a ratio *alpha* / *beta* = *w*, such that $A - w*B$ is singular. It is usually represented as the pair (*alpha, beta*), as there is a reasonable interpretation for *beta*=0 or for both being zero.
A pair of matrices (*S,T*) is in generalized real Schur form if *T* is upper triangular with non-negative diagonal and *S* is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond

to real generalized eigenvalues, while 2-by-2 blocks of *S* will be “standardized" by making the corresponding elements of *T* have the form:

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$$

and the pair of corresponding 2-by-2 blocks in *S* and *T* will have a complex conjugate pair of generalized eigenvalues.
A pair of matrices (*S,T*) is in generalized complex Schur form if *S* and *T* are upper triangular and, in addition, the diagonal of *T* are non-negative real numbers.

Input Parameters

jobvsl — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobvsl* = `'N'`, then the left Schur vectors are not computed.
If *jobvsl* = `'V'`, then the left Schur vectors are computed.

jobvsr — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobvsr* = `'N'`, then the right Schur vectors are not computed.
If *jobvsr* = `'V'`, then the right Schur vectors are computed.

sort — `CHARACTER*1`. Must be `'N'` or `'S'`.
Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form.

If *sort* = `'N'`, then eigenvalues are not ordered.
If *sort* = `'S'`, eigenvalues are ordered (see *selctg*).

selctg — `LOGICAL FUNCTION` of three `REAL` arguments for real flavors.
`LOGICAL FUNCTION` of two `COMPLEX` arguments for complex flavors.

selctg must be declared `EXTERNAL` in the calling subroutine.
If *sort* = `'S'`, *selctg* is used to select eigenvalues to sort to the top left of the Schur form.
If *sort* = `'N'`, *selctg* is not referenced.

For real flavors:
An eigenvalue (*alphar*(j) + *alphai*(j))/*beta*(j) is selected if *selctg*(*alphar*(j), *alphai*(j), *beta*(j)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.
Note that in the ill-conditioned case, a selected complex eigenvalue may no

longer satisfy
selctg(*alphar*(j), *alphai*(j), *beta*(j)) = .TRUE. after ordering. In this case *info* is set to *n*+2 .

For complex flavors:
An eigenvalue *alpha*(j) / *beta*(j) is selected if *selctg*(*alpha*(j), *beta*(j)) is true.
Note that a selected complex eigenvalue may no longer satisfy *selctg*(*alpha*(j), *beta*(j)) = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case *info* is set to *n*+2 (see *info* below).

sense — CHARACTER*1. Must be 'N', 'E', 'V', or 'B'.
Determines which reciprocal condition number are computed.

If *sense* ='N', none are computed;
If *sense* ='E', computed for average of selected eigenvalues only;
If *sense* ='V', computed for selected deflating subspaces only;
If *sense* ='B', computed for both.
If *sense* is 'E', 'V', or 'B', then *sort* must equal 'S'.

n — INTEGER. The order of the matrices *A*, *B*, *vsl*, and *vsr* ($n \geq 0$).

a, b, work — REAL for sggesx
DOUBLE PRECISION for dggesx
COMPLEX for cggesx
DOUBLE COMPLEX for zggesx.
Arrays:
a(*lda*,*) is an array containing the *n*-by-*n* matrix *A* (first of the pair of matrices).
The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) is an array containing the *n*-by-*n* matrix *B* (second of the pair of matrices).
The second dimension of *b* must be at least max(1, *n*).

work(*lwork*) is a workspace array.

lda — INTEGER. The first dimension of the array *a*.
Must be at least max(1, *n*) .

ldb — INTEGER. The first dimension of the array *b*.
Must be at least max(1, *n*) .

ldvsl, ldvsr — `INTEGER`. The first dimensions of the output matrices *vsl* and *vsr*, respectively. Constraints:
ldvsl ≥ 1. If *jobvsl* = 'V', *ldvsl* ≥ max(1, *n*).
ldvsr ≥ 1. If *jobvsr* = 'V', *ldvsr* ≥ max(1, *n*).

lwork — `INTEGER`. The dimension of the array *work*.
For real flavors:
lwork ≥ max(1, 8(*n*+1)+16);
if *sense* = 'E', 'V', or 'B', then
lwork ≥ max(8(*n*+1)+16), 2**sdim**(*n*-*sdim*)).
For complex flavors:
lwork ≥ max(1, 2*n*);
if *sense* = 'E', 'V', or 'B', then
lwork ≥ max(2*n*, 2**sdim**(*n*-*sdim*)).

For good performance, *lwork* must generally be larger.

rwork — `REAL` for `cggesx`
`DOUBLE PRECISION` for `zggesx`
Workspace array, `DIMENSION` at least max(1, 8*n*).
This array is used in complex flavors only.

iwork — `INTEGER`.
Workspace array, `DIMENSION` (*liwork*). Not referenced if *sense* = 'N'.

liwork — `INTEGER`. The dimension of the array *iwork*.

liwork ≥ *n*+6 for real flavors;
liwork ≥ *n*+2 for complex flavors.

bwork — `LOGICAL`.
Workspace array, `DIMENSION` at least max(1, *n*).
Not referenced if *sort* = 'N'.

Output Parameters

a — On exit, this array has been overwritten by its generalized Schur form S.

b — On exit, this array has been overwritten by its generalized Schur form T.

sdim — `INTEGER`.
If *sort* = 'N', *sdim*= 0.
If *sort* = 'S', *sdim* is equal to the number of eigenvalues (after sorting) for which *selctg* is true.
Note that for real flavors complex conjugate pairs for which *selctg* is true for either eigenvalue count as 2.

alphar,alphai REAL for sggesx;
DOUBLE PRECISION for dggesx.
Arrays, DIMENSION at least max(1,*n*) each. Contain values that form generalized eigenvalues in real flavors.
See *beta*.

alpha COMPLEX for cggesx;
DOUBLE COMPLEX for zggesx.
Array, DIMENSION at least max(1,*n*). Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta REAL for sggesx
DOUBLE PRECISION for dggesx
COMPLEX for cggesx
DOUBLE COMPLEX for zggesx.
Array, DIMENSION at least max(1,*n*).
For real flavors:
On exit, (*alphar*(j) + *alphai*(j)*i)/*beta*(j), j=1,...,*n*, will be the generalized eigenvalues.
alphar(j) + *alphai*(j)*i and *beta*(j), j=1,...,*n* are the diagonals of the complex Schur form (*S,T*) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (*A,B*) were further reduced to triangular form using complex unitary transformations. If *alphai*(j) is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with *alphai*(j+1) negative.
For complex flavors:
On exit, *alpha*(j)/*beta*(j), j=1,...,*n*, will be the generalized eigenvalues.
alpha(j), j=1,...,*n*, and *beta*(j), j=1,...,*n*, are the diagonals of the complex Schur form (*S,T*) output by cggesx/zggesx. The *beta*(j) will be non-negative real.

See also *Application Notes* below.

vsl, vsr REAL for sggesx
DOUBLE PRECISION for dggesx
COMPLEX for cggesx
DOUBLE COMPLEX for zggesx.
Arrays:
vsl(*ldvsl*,*), the second dimension of *vsl* must be at least max(1, *n*).
If *jobvsl* ='V', this array will contain the left Schur vectors.
If *jobvsl* ='N', *vsl* is not referenced.

	vsr(*ldvsr*,*), the second dimension of *vsr* must be at least max(1, *n*). If *jobvsr* ='V', this array will contain the right Schur vectors. If *jobvsr* ='N', *vsr* is not referenced.
rconde, *rcondv*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION (2) each If *sense* = 'E' or 'B' , *rconde*(1) and *rconde*(2) contain the reciprocal condition numbers for the average of the selected eigenvalues. Not referenced if *sense* = 'N' or 'V'. If *sense* = 'V' or 'B' , *rcondv*(1) and *rcondv*(2) contain the reciprocal condition numbers for the selected deflating subspaces. Not referenced if *sense* = 'N' or 'E'.
work(1)	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *-i*, the *i*th parameter had an illegal value. If *info* = *i*, and $i \le n$: the *QZ* iteration failed. (*A,B*) is not in Schur form, but *alphar*(j), *alphai*(j) (for real flavors), or *alpha*(j) (for complex flavors), and *beta*(j), j=*info*+1,...,*n* should be correct. $i > n$: errors that usually indicate LAPACK problems: *i* = *n*+1: other than *QZ* iteration failed in ?hgeqz; *i* = *n*+2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the generalized Schur form no longer satisfy *selctg* = .TRUE.. This could also be caused due to scaling; *i* = *n*+3: reordering failed in ?tgsen.

Application Notes

If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The quotients `alphar`(j)/`beta`(j) and `alphai`(j)/`beta`(j) may easily over- or underflow, and `beta`(j) may even be zero. Thus, you should avoid simply computing the ratio. However, `alphar` and `alphai` will be always less than and usually comparable with norm(*A*) in magnitude, and `beta` always less than and usually comparable with norm(*B*).

?ggev

Computes the generalized eigenvalues, and the left and/or right generalized eigenvectors for a pair of nonsymmetric matrices.

Syntax

```
call sggev ( jobvl, jobvr, n, a, lda, b, ldb, alphar, alphai, beta,
            vl, ldvl, vr, ldvr, work, lwork, info )
call dggev ( jobvl, jobvr, n, a, lda, b, ldb, alphar, alphai, beta,
            vl, ldvl, vr, ldvr, work, lwork, info )
call cggev ( jobvl, jobvr, n, a, lda, b, ldb, alpha, beta,
            vl, ldvl, vr, ldvr, work, lwork, rwork, info )
call zggev ( jobvl, jobvr, n, a, lda, b, ldb, alpha, beta,
            vl, ldvl, vr, ldvr, work, lwork, rwork, info )
```

Description

This routine computes for a pair of *n*-by-*n* real/complex nonsymmetric matrices (*A,B*), the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

A generalized eigenvalue for a pair of matrices (*A,B*) is a scalar λ or a ratio *alpha* / *beta* = λ, such that $A - \lambda * B$ is singular. It is usually represented as the pair (*alpha, beta*), as there is a reasonable interpretation for *beta*=0 and even for both being zero.
The right generalized eigenvector $v(j)$ corresponding to the generalized eigenvalue $\lambda(j)$ of (*A,B*) satisfies

$$A * v(j) = \lambda(j) * B * v(j) .$$

The left generalized eigenvector $u(j)$ corresponding to the generalized eigenvalue $\lambda(j)$ of (*A,B*) satisfies

$$u(j)^H * A = \lambda(j) * u(j)^H * B$$

where $u(j)^H$ denotes the conjugate transpose of $u(j)$.

Input Parameters

jobvl — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobvl* = `'N'`, the left generalized eigenvectors are not computed;
If *jobvl* = `'V'`, the left generalized eigenvectors are computed.

jobvr	`CHARACTER*1`. Must be `'N'` or `'V'`. If *jobvr* = `'N'`, the right generalized eigenvectors are not computed; If *jobvr* = `'V'`, the right generalized eigenvectors are computed.
n	`INTEGER`. The order of the matrices *A*, *B*, *vl*, and *vr* ($n \geq 0$).
a, b, work	`REAL` for `sggev` `DOUBLE PRECISION` for `dggev` `COMPLEX` for `cggev` `DOUBLE COMPLEX` for `zggev`. Arrays: *a*(*lda*,*) is an array containing the *n*-by-*n* matrix *A* (first of the pair of matrices). The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) is an array containing the *n*-by-*n* matrix *B* (second of the pair of matrices). The second dimension of *b* must be at least max(1, *n*). *work*(*lwork*) is a workspace array.
lda	`INTEGER`. The first dimension of the array *a*. Must be at least max(1, *n*) .
ldb	`INTEGER`. The first dimension of the array *b*. Must be at least max(1, *n*) .
ldvl, *ldvr*	`INTEGER`. The first dimensions of the output matrices *vl* and *vr*, respectively. Constraints: *ldvl* ≥ 1. If *jobvl* = `'V'`, *ldvl* ≥ max(1, *n*) . *ldvr* ≥ 1. If *jobvr* = `'V'`, *ldvr* ≥ max(1, *n*) .
lwork	`INTEGER`. The dimension of the array *work*. *lwork* ≥ max(1, 8*n*+16) for real flavors; *lwork* ≥ max(1, 2*n*) for complex flavors. For good performance, *lwork* must generally be larger.
rwork	`REAL` for `cggev` `DOUBLE PRECISION` for `zggev` Workspace array, `DIMENSION` at least max(1, 8*n*). This array is used in complex flavors only.

Output Parameters

a, b	On exit, these arrays have been overwritten.

alphar,alphai REAL for sggev;
DOUBLE PRECISION for dggev.
Arrays, DIMENSION at least max(1,*n*) each. Contain values that form generalized eigenvalues in real flavors.
See *beta*.

alpha COMPLEX for cggev;
DOUBLE COMPLEX for zggev.
Array, DIMENSION at least max(1,*n*). Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta REAL for sggev
DOUBLE PRECISION for dggev
COMPLEX for cggev
DOUBLE COMPLEX for zggev.
Array, DIMENSION at least max(1,*n*).
For real flavors:
On exit, (*alphar*(j) + *alphai*(j)*i)/*beta*(j), j=1,...,*n*, will be the generalized eigenvalues.
If *alphai*(j) is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with *alphai*(j+1) negative.
For complex flavors:
On exit, *alpha*(j)/*beta*(j), j=1,...,*n*, will be the generalized eigenvalues.

See also *Application Notes* below.

vl, vr REAL for sggev
DOUBLE PRECISION for dggev
COMPLEX for cggev
DOUBLE COMPLEX for zggev.
Arrays:
vl(*ldvl*,*); the second dimension of *vl* must be at least max(1, *n*).

If *jobvl* ='V', the left generalized eigenvectors *u*(j) are stored one after another in the columns of *vl*, in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have abs(Re) + abs(Im) = 1. If *jobvl* ='N', *vl* is not referenced.
For real flavors:
If the j-th eigenvalue is real, then *u*(j) = *vl*(:,j), the j-th column of *vl*. If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then *u*(j) = *vl*(:,j) + *i***vl*(:,j+1) and *u*(j+1) = *vl*(:,j) - *i***vl*(:,j+1), where $i=\sqrt{-1}$.

For complex flavors:
u(j) = `vl`(:,j), the j-th column of `vl`.

`vr(ldvr,*)`; the second dimension of `vr` must be at least max(1, *n*).

If `jobvr` = 'V', the right generalized eigenvectors v(j) are stored one after another in the columns of `vr`, in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have abs(Re) + abs(Im) = 1. If `jobvr` = 'N', `vr` is not referenced.
For real flavors:
If the j-th eigenvalue is real, then v(j) = `vr`(:,j), the j-th column of `vr`. If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then v(j) = `vr`(:,j) + *i**`vr`(:,j+1) and v(j+1) = `vr`(:,j) - *i**`vr`(:,j+1).

For complex flavors:
v(j) = `vr`(:,j), the j-th column of `vr`.

`work(1)` On exit, if `info` = 0, then `work(1)` returns the required minimal size of `lwork`.

`info` INTEGER.
If `info` = 0, the execution is successful.
If `info` = `-i`, the `i`th parameter had an illegal value.
If `info` = `i`, and
$i \le n$:

the *QZ* iteration failed. No eigenvectors have been calculated, but `alphar`(j), `alphai`(j) (for real flavors), or `alpha`(j) (for complex flavors), and `beta`(j), j=`info`+1,...,*n* should be correct.

$i > n$: errors that usually indicate LAPACK problems:

$i = n+1$: other than *QZ* iteration failed in ?hgeqz;

$i = n+2$: error return from ?tgevc.

Application Notes

If you are in doubt how much workspace to supply for the array `work`, use a generous value of `lwork` for the first run. On exit, examine `work(1)` and use this value for subsequent runs.

The quotients `alphar`(j)/`beta`(j) and `alphai`(j)/`beta`(j) may easily over- or underflow, and `beta`(j) may even be zero. Thus, you should avoid simply computing the ratio. However, `alphar` and `alphai` (for real flavors) or `alpha` (for complex flavors) will be always less than and usually comparable with norm(*A*) in magnitude, and `beta` always less than and usually comparable with norm(*B*).

?ggevx

Computes the generalized eigenvalues, and, optionally, the left and/or right generalized eigenvectors.

Syntax

```
call sggevx ( balanc, jobvl, jobvr, sense, n, a, lda, b, ldb,
              alphar, alphai, beta, vl, ldvl, vr, ldvr, ilo, ihi,
              lscale, rscale, abnrm, bbnrm, rconde, rcondv, work,
               lwork, iwork, bwork, info)
call dggevx ( balanc, jobvl, jobvr, sense, n, a, lda, b, ldb,
              alphar, alphai, beta, vl, ldvl, vr, ldvr, ilo, ihi,
              lscale, rscale, abnrm, bbnrm, rconde, rcondv, work,
               lwork, iwork, bwork, info)
call cggevx ( balanc, jobvl, jobvr, sense, n, a, lda, b, ldb,
              alpha, beta, vl, ldvl, vr, ldvr, ilo, ihi,
              lscale, rscale, abnrm, bbnrm, rconde, rcondv, work,
               lwork, rwork, iwork, bwork, info)
call zggevx ( balanc, jobvl, jobvr, sense, n, a, lda, b, ldb,
              alpha, beta, vl, ldvl, vr, ldvr, ilo, ihi,
              lscale, rscale, abnrm, bbnrm, rconde, rcondv, work,
               lwork, rwork, iwork, bwork, info)
```

Description

This routine computes for a pair of *n*-by-*n* real/complex nonsymmetric matrices (*A,B*), the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (*ilo*, *ihi*, *lscale*, *rscale*, *abnrm*, and *bbnrm*), reciprocal condition numbers for the eigenvalues (*rconde*), and reciprocal condition numbers for the right eigenvectors (*rcondv*).

A generalized eigenvalue for a pair of matrices (*A,B*) is a scalar λ or a ratio $alpha / beta = \lambda$, such that $A - \lambda * B$ is singular. It is usually represented as the pair (*alpha, beta*), as there is a reasonable interpretation for *beta*=0 and even for both being zero.
The right generalized eigenvector *v*(j) corresponding to the generalized eigenvalue λ(j) of (*A,B*) satisfies

$$A * v(j) = \lambda(j) * B * v(j) .$$

The left generalized eigenvector u(j) corresponding to the generalized eigenvalue λ(j) of (A,B) satisfies

$$u(j)^H * A = \lambda(j) * u(j)^H * B$$

where $u(j)^H$ denotes the conjugate transpose of u(j).

Input Parameters

balanc — `CHARACTER*1`. Must be `'N'`, `'P'`, `'S'`, or `'B'`.
Specifies the balance option to be performed.

If *balanc* = `'N'`, do not diagonally scale or permute;
If *balanc* = `'P'`, permute only;
If *balanc* = `'S'`, scale only;
If *balanc* = `'B'`, both permute and scale.

Computed reciprocal condition numbers will be for the matrices after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.

jobvl — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobvl* = `'N'`, the left generalized eigenvectors are not computed;
If *jobvl* = `'V'`, the left generalized eigenvectors are computed.

jobvr — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobvr* = `'N'`, the right generalized eigenvectors are not computed;
If *jobvr* = `'V'`, the right generalized eigenvectors are computed.

sense — `CHARACTER*1`. Must be `'N'`, `'E'`, `'V'`, or `'B'`.
Determines which reciprocal condition number are computed.

If *sense* = `'N'`, none are computed;
If *sense* = `'E'`, computed for eigenvalues only;
If *sense* = `'V'`, computed for eigenvectors only;
If *sense* = `'B'`, computed for eigenvalues and eigenvectors.

n — `INTEGER`. The order of the matrices A, B, *vl*, and *vr* ($n \ge 0$).

a, b, work — `REAL` for `sggevx`
`DOUBLE PRECISION` for `dggevx`
`COMPLEX` for `cggevx`
`DOUBLE COMPLEX` for `zggevx`.
Arrays:

a(*lda*,*) is an array containing the *n*-by-*n* matrix *A* (first of the pair of matrices).
The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) is an array containing the *n*-by-*n* matrix *B* (second of the pair of matrices).
The second dimension of *b* must be at least max(1, *n*).

work(*lwork*) is a workspace array.

lda — INTEGER. The first dimension of the array *a*.
Must be at least max(1, *n*) .

ldb — INTEGER. The first dimension of the array *b*.
Must be at least max(1, *n*) .

ldvl, *ldvr* — INTEGER. The first dimensions of the output matrices *vl* and *vr*, respectively.
Constraints:
ldvl $\geq$ 1. If *jobvl* = 'V', *ldvl* $\geq$ max(1, *n*) .
ldvr $\geq$ 1. If *jobvr* = 'V', *ldvr* $\geq$ max(1, *n*) .

lwork — INTEGER. The dimension of the array *work*.
For real flavors:
lwork $\geq$ max(1, 6*n*);
if *sense* = 'E', *lwork* $\geq$ 12*n* ;
if *sense* = 'V', or 'B', *lwork* $\geq$ $2n^2 + 12n + 16$.
For complex flavors:
lwork $\geq$ max(1, 2*n*);
if *sense* = 'N', or 'E', *lwork* $\geq$ 2*n* ;
if *sense* = 'V', or 'B', *lwork* $\geq$ $2n^2 + 2n$.

rwork — REAL for cggevx
DOUBLE PRECISION for zggevx
Workspace array, DIMENSION at least max(1, 6*n*).
This array is used in complex flavors only.

iwork — INTEGER.
Workspace array, DIMENSION at least (*n*+6) for real flavors and at least (*n*+2) for complex flavors.
Not referenced if *sense* = 'E'.

bwork — LOGICAL.
Workspace array, DIMENSION at least max(1, *n*).
Not referenced if *sense* = 'N'.

Output Parameters

a, b

On exit, these arrays have been overwritten.

If *jobvl* = 'V' or *jobvr* = 'V' or both, then *a* contains the first part of the real Schur form of the "balanced" versions of the input *A* and *B*, and *b* contains its second part.

alphar, alphai

REAL for sggevx;
DOUBLE PRECISION for dggevx.
Arrays, DIMENSION at least max(1,*n*) each. Contain values that form generalized eigenvalues in real flavors.
See *beta*.

alpha

COMPLEX for cggevx;
DOUBLE COMPLEX for zggevx.
Array, DIMENSION at least max(1,*n*). Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta

REAL for sggevx
DOUBLE PRECISION for dggevx
COMPLEX for cggevx
DOUBLE COMPLEX for zggevx.
Array, DIMENSION at least max(1,*n*).
For real flavors:
On exit, (*alphar*(j) + *alphai*(j)*i)/*beta*(j), j=1,...,*n*, will be the generalized eigenvalues.
If *alphai*(j) is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with *alphai*(j+1) negative.
For complex flavors:
On exit, *alpha*(j)/*beta*(j), j=1,...,*n*, will be the generalized eigenvalues.

See also *Application Notes* below.

vl, vr

REAL for sggevx
DOUBLE PRECISION for dggevx
COMPLEX for cggevx
DOUBLE COMPLEX for zggevx.
Arrays:
vl(*ldvl*,*); the second dimension of *vl* must be at least max(1, *n*).

If *jobvl* = 'V', the left generalized eigenvectors *u*(j) are stored one after another in the columns of *vl*, in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have abs(Re) + abs(Im) =

1. If `jobvl` = 'N', `vl` is not referenced.
For real flavors:
If the j-th eigenvalue is real, then *u*(j) = `vl`(:,j), the j-th column of `vl`. If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then *u*(j) = `vl`(:,j) + *i**`vl`(:,j+1) and *u*(j+1) = `vl`(:,j) - *i**`vl`(:,j+1), where $i = \sqrt{-1}$.

For complex flavors:
u(j) = `vl`(:,j), the j-th column of `vl`.

`vr(ldvr,*)`; the second dimension of `vr` must be at least max(1, *n*).

If `jobvr` = 'V', the right generalized eigenvectors *v*(j) are stored one after another in the columns of `vr`, in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have abs(Re) + abs(Im) = 1. If `jobvr` = 'N', `vr` is not referenced.
For real flavors:
If the j-th eigenvalue is real, then *v*(j) = `vr`(:,j), the j-th column of `vr`. If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then *v*(j) = `vr`(:,j) + *i**`vr`(:,j+1) and *v*(j+1) = `vr`(:,j) - *i**`vr`(:,j+1).

For complex flavors:
v(j) = `vr`(:,j), the j-th column of `vr`.

`ilo, ihi` INTEGER.
`ilo` and `ihi` are integer values such that on exit
A(i,j) = 0 and *B*(i,j) = 0 if i > j and j = 1,..., `ilo`-1 or
i = `ihi`+1,..., *n*.
If `balanc` = 'N' or 'S', `ilo` = 1 and `ihi` = *n*.

`lscale,rscale` REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Arrays, DIMENSION at least max(1, *n*) each.
`lscale` contains details of the permutations and scaling factors applied to the left side of *A* and *B*.
If *PL*(j) is the index of the row interchanged with row j, and *DL*(j) is the scaling factor applied to row j, then

`lscale`(j) = *PL*(j), for j = 1,...,`ilo`-1

= *DL*(j), for j = `ilo`,...,`ihi`

= *PL*(j) for j = `ihi`+1,...,*n*.

The order in which the interchanges are made is *n* to `ihi`+1, then 1 to `ilo`-1.

rscale contains details of the permutations and scaling factors applied to the right side of *A* and *B*.
If *PR*(j) is the index of the column interchanged with column j, and *DR*(j) is the scaling factor applied to column j, then

rscale(j) = *PR*(j), for j = 1,...,*ilo*-1

= *DR*(j), for j = *ilo*,...,*ihi*

= *PR*(j) for j = *ihi*+1,...,*n*.

The order in which the interchanges are made is *n* to *ihi*+1, then 1 to *ilo*-1.

abnrm, bbnrm — REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.

The one-norms of the balanced matrices *A* and *B*, respectively.

rconde, rcondv — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(1, *n*) each.

If *sense* = 'E', or 'B', *rconde* contains the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array. For a complex conjugate pair of eigenvalues two consecutive elements of *rconde* are set to the same value. Thus *rconde*(j), *rcondv*(j), and the j-th columns of *vl* and *vr* all correspond to the same eigenpair (but not in general the j-th eigenpair, unless all eigenpairs are selected).
If *sense* = 'V', *rconde* is not referenced.

If *sense* = 'V', or 'B', *rcondv* contains the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array. For a complex eigenvector two consecutive elements of *rcondv* are set to the same value. If the eigenvalues cannot be reordered to compute *rcondv*(j), *rcondv*(j) is set to 0; this can only occur when the true value would be very small anyway.
If *sense* = 'E', *rcondv* is not referenced.

work(1) — On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.
If *info* = *i*, and
$i \le n$:

the *QZ* iteration failed. No eigenvectors have been calculated, but *alphar*(j), *alphai*(j) (for real flavors), or *alpha*(j) (for complex flavors), and *beta*(j), j=*info*+1,...,*n* should be correct.

i > *n* : errors that usually indicate LAPACK problems:

i = *n*+1: other than *QZ* iteration failed in ?hgeqz;

i = *n*+2: error return from ?tgevc.

Application Notes

If you are in doubt how much workspace to supply for the array *work*, use a generous value of *lwork* for the first run. On exit, examine *work*(1) and use this value for subsequent runs.

The quotients *alphar*(j)/*beta*(j) and *alphai*(j)/*beta*(j) may easily over- or underflow, and *beta*(j) may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* (for real flavors) or *alpha* (for complex flavors) will be always less than and usually comparable with norm(*A*) in magnitude, and *beta* always less than and usually comparable with norm(*B*).

LAPACK Auxiliary and Utility Routines

5

This chapter describes the Intel® Math Kernel Library implementation of LAPACK auxiliary and utility routines. The library includes auxiliary routines for both real and complex data.

Auxiliary Routines

Routine naming conventions, mathematical notation, and matrix storage schemes used for LAPACK auxiliary routines are the same as for the driver and computational routines described in previous chapters.

The table below summarizes information about the available LAPACK auxiliary routines.

Table 5-1 LAPACK Auxiliary Routines

Routine Name	Data Types	Description
?lacgv	c, z	Conjugates a complex vector.
?lacrm	c, z	Multiplies a complex matrix by a square real matrix.
?lacrt	c, z	Performs a linear transformation of a pair of complex vectors.
?laesy	c, z	Computes the eigenvalues and eigenvectors of a 2-by-2 complex symmetric matrix.
?rot	c, z	Applies a plane rotation with real cosine and complex sine to a pair of complex vectors.
?spmv	c, z	Computes a matrix-vector product for complex vectors using a complex symmetric packed matrix
?spr	c, z	Performs the symmetrical rank-1 update of a complex symmetric packed matrix.
?symv	c, z	Computes a matrix-vector product for a complex symmetric matrix.

Table 5-1 LAPACK Auxiliary Routines (continued)

Routine Name	Data Types	Description
?syr	c, z	Performs the symmetric rank-1 update of a complex symmetric matrix.
i?max1	c, z	Finds the index of the vector element whose real part has maximum absolute value.
?sum1	sc,dz	Forms the 1-norm of the complex vector using the true absolute value.
?gbtf2	s,d,c,z	Computes the LU factorization of a general band matrix using the unblocked version of the algorithm.
?gebd2	s,d,c,z	Reduces a general matrix to bidiagonal form using an unblocked algorithm.
?gehd2	s,d,c,z	Reduces a general square matrix to upper Hessenberg form using an unblocked algorithm.
?gelq2	s,d,c,z	Computes the LQ factorization of a general rectangular matrix using an unblocked algorithm.
?geql2	s,d,c,z	Computes the QL factorization of a general rectangular matrix using an unblocked algorithm.
?geqr2	s,d,c,z	Computes the QR factorization of a general rectangular matrix using an unblocked algorithm.
?gerq2	s,d,c,z	Computes the RQ factorization of a general rectangular matrix using an unblocked algorithm.
?gesc2	s,d,c,z	Solves a system of linear equations using the LU factorization with complete pivoting computed by `?getc2`.
?getc2	s,d,c,z	Computes the LU factorization with complete pivoting of the general n-by-n matrix.
?getf2	s,d,c,z	Computes the LU factorization of a general m by n matrix using partial pivoting with row interchanges (unblocked algorithm).
?gtts2	s,d,c,z	Solves a system of linear equations with a tridiagonal matrix using the LU factorization computed by `?gttrf`.
?labrd	s,d,c,z	Reduces the first *nb* rows and columns of a general matrix to a bidiagonal form.
?lacon	s,d,c,z	Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.
?lacpy	s,d,c,z	Copies all or part of one two-dimensional array to another.
?ladiv	s,d,c,z	Performs complex division in real arithmetic, avoiding unnecessary overflow.

Table 5-1 LAPACK Auxiliary Routines (continued)

Routine Name	Data Types	Description
`?lae2`	`s,d`	Computes the eigenvalues of a 2-by-2 symmetric matrix.
`?laebz`	`s,d`	Computes the number of eigenvalues of a real symmetric tridiagonal matrix which are less than or equal to a given value, and performs other tasks required by the routine `?stebz`.
`?laed0`	`s,d,c,z`	Used by `?stedc`. Computes all eigenvalues and corresponding eigenvectors of an unreduced symmetric tridiagonal matrix using the divide and conquer method.
`?laed1`	`s,d`	Used by `sstedc/dstedc`. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is tridiagonal.
`?laed2`	`s,d`	Used by `sstedc/dstedc`. Merges eigenvalues and deflates secular equation. Used when the original matrix is tridiagonal.
`?laed3`	`s,d`	Used by `sstedc/dstedc`. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is tridiagonal.
`?laed4`	`s,d`	Used by `sstedc/dstedc`. Finds a single root of the secular equation.
`?laed5`	`s,d`	Used by `sstedc/dstedc`. Solves the 2-by-2 secular equation.
`?laed6`	`s,d`	Used by `sstedc/dstedc`. Computes one Newton step in solution of the secular equation.
`?laed7`	`s,d,c,z`	Used by `?stedc`. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is dense.
`?laed8`	`s,d,c,z`	Used by `?stedc`. Merges eigenvalues and deflates secular equation. Used when the original matrix is dense.
`?laed9`	`s,d`	Used by `sstedc/dstedc`. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is dense.
`?laeda`	`s,d`	Used by `?stedc`. Computes the Z vector determining the rank-one modification of the diagonal matrix. Used when the original matrix is dense.
`?laein`	`s,d,c,z`	Computes a specified right or left eigenvector of an upper Hessenberg matrix by inverse iteration.
`?laev2`	`s,d,c,z`	Computes the eigenvalues and eigenvectors of a 2-by-2 symmetric/Hermitian matrix.

Table 5-1 LAPACK Auxiliary Routines (continued)

Routine Name	Data Types	Description
?laexc	s,d	Swaps adjacent diagonal blocks of a real upper quasi-triangular matrix in Schur canonical form, by an orthogonal similarity transformation.
?lag2	s,d	Computes the eigenvalues of a 2-by-2 generalized eigenvalue problem, with scaling as necessary to avoid over-/underflow.
?lags2	s,d	Computes 2-by-2 orthogonal matrices U, V, and Q, and applies them to matrices A and B such that the rows of the transformed A and B are parallel.
?lagtf	s,d	Computes an LU factorization of a matrix $T-\lambda I$, where T is a general tridiagonal matrix, and λ a scalar, using partial pivoting with row interchanges.
?lagtm	s,d,c,z	Performs a matrix-matrix product of the form $C = \alpha AB+\beta C$, where A is a tridiagonal matrix, B and C are rectangular matrices, and α and β are scalars, which may be 0, 1, or -1.
?lagts	s,d	Solves the system of equations $(T-\lambda I)x = y$ or $(T-\lambda I)^T x = y$,where T is a general tridiagonal matrix and λ a scalar, using the LU factorization computed by ?lagtf.
?lagv2	s,d	Computes the Generalized Schur factorization of a real 2-by-2 matrix pencil (A,B) where B is upper triangular.
?lahqr	s,d,c,z	Computes the eigenvalues and Schur factorization of an upper Hessenberg matrix, using the double-shift/single-shift QR algorithm.
?lahrd	s,d,c,z	Reduces the first nb columns of a general rectangular matrix A so that elements below the k-th subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A.
?laic1	s,d,c,z	Applies one step of incremental condition estimation.
?laln2	s,d	Solves a 1-by-1 or 2-by-2 linear system of equations of the specified form.
?lals0	s,d,c,z	Applies back multiplying factors in solving the least squares problem using divide and conquer SVD approach. Used by ?gelsd.
?lalsa	s,d,c,z	Computes the SVD of the coefficient matrix in compact form. Used by ?gelsd.
?lalsd	s,d,c,z	Uses the singular value decomposition of A to solve the least squares problem.

Table 5-1 LAPACK Auxiliary Routines (continued)

Routine Name	Data Types	Description
?lamrg	s,d	Creates a permutation list to merge the entries of two independently sorted sets into a single set sorted in acsending order.
?langb	s,d,c,z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of general band matrix.
?lange	s,d,c,z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general rectangular matrix.
?langt	s,d,c,z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general tridiagonal matrix.
?lanhs	s,d,c,z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of an upper Hessenberg matrix.
?lansb	s,d,c,z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric band matrix.
?lanhb	c,z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hermitian band matrix.
?lansp	s,d,c,z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric matrix supplied in packed form.
?lanhp	c,z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix supplied in packed form.
?lanst/?lanht	s,d/c,z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real symmetric or complex Hermitian tridiagonal matrix.
?lansy	s,d,c,z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix.
?lanhe	c,z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix.

Table 5-1 LAPACK Auxiliary Routines (continued)

Routine Name	Data Types	Description
?lantb	s,d,c,z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular band matrix.
?lantp	s,d,c,z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular matrix supplied in packed form.
?lantr	s,d,c,z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix.
?lanv2	s,d	Computes the Schur factorization of a real 2-by-2 nonsymmetric matrix in standard form.
?lapll	s,d,c,z	Measures the linear dependence of two vectors.
?lapmt	s,d,c,z	Performs a forward or backward permutation of the columns of a matrix.
?lapy2	s,d	Returns $\mathrm{sqrt}(x^2+y^2)$.
?lapy3	s,d	Returns $\mathrm{sqrt}(x^2+y^2+z^2)$.
?laqgb	s,d,c,z	Scales a general band matrix, using row and column scaling factors computed by ?gbequ.
?laqge	s,d,c,z	Scales a general rectangular matrix, using row and column scaling factors computed by ?geequ.
?laqp2	s,d,c,z	Computes a QR factorization with column pivoting of the matrix block.
?laqps	s,d,c,z	Computes a step of QR factorization with column pivoting of a real m-by-n matrix A by using BLAS level 3.
?laqsb	s,d,c,z	Scales a symmetric/Hermitian band matrix, using scaling factors computed by ?pbequ.
?laqsp	s,d,c,z	Scales a symmetric/Hermitian matrix in packed storage, using scaling factors computed by ?ppequ.
?laqsy	s,d,c,z	Scales a symmetric/Hermitian matrix, using scaling factors computed by ?poequ.
?laqtr	s,d	Solves a real quasi-triangular system of equations, or a complex quasi-triangular system of special form, in real arithmetic.
?lar1v	s,d,c,z	Computes the (scaled) *r*-th column of the inverse of the submatrix in rows *b1* through *bn* of the tridiagonal matrix $LDL^T - \sigma I$.

Table 5-1 LAPACK Auxiliary Routines (continued)

Routine Name	Data Types	Description
?lar2v	s,d,c,z	Applies a vector of plane rotations with real cosines and real/complex sines from both sides to a sequence of 2-by-2 symmetric/Hermitian matrices.
?larf	s,d,c,z	Applies an elementary reflector to a general rectangular matrix.
?larfb	s,d,c,z	Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.
?larfg	s,d,c,z	Generates an elementary reflector (Householder matrix).
?larft	s,d,c,z	Forms the triangular factor T of a block reflector $H = I - VTV^H$
?larfx	s,d,c,z	Applies an elementary reflector to a general rectangular matrix, with loop unrolling when the reflector has order $\leq$ 10.
?largv	s,d,c,z	Generates a vector of plane rotations with real cosines and real/complex sines.
?larnv	s,d,c,z	Returns a vector of random numbers from a uniform or normal distribution.
?larrb	s,d	Provides limited bisection to locate eigenvalues for more accuracy.
?larre	s,d	Given the tridiagonal matrix T, sets small off-diagonal elements to zero and for each unreduced block T_i, finds base representations and eigenvalues.
?larrf	s,d	Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.
?larrv	s,d,c,z	Computes the eigenvectors of the tridiagonal matrix $T = L D L^T$ given L, D and the eigenvalues of $L D L^T$.
?lartg	s,d,c,z	Generates a plane rotation with real cosine and real/complex sine.
?lartv	s,d,c,z	Applies a vector of plane rotations with real cosines and real/complex sines to the elements of a pair of vectors.
?laruv	s,d	Returns a vector of n random real numbers from a uniform distribution.
?larz	s,d,c,z	Applies an elementary reflector (as returned by ?tzrzf) to a general matrix.
?larzb	s,d,c,z	Applies a block reflector or its transpose/conjugate-transpose to a general matrix.
?larzt	s,d,c,z	Forms the triangular factor T of a block reflector $H = I - VTV^H$.
?las2	s,d	Computes singular values of a 2-by-2 triangular matrix.
?lascl	s,d,c,z	Multiplies a general rectangular matrix by a real scalar defined as c_{to}/c_{from}.

Table 5-1 LAPACK Auxiliary Routines (continued)

Routine Name	Data Types	Description
?lasd0	s,d	Computes the singular values of a real upper bidiagonal n-by-m matrix B with diagonal *d* and off-diagonal *e*. Used by ?bdsdc.
?lasd1	s,d	Computes the SVD of an upper bidiagonal matrix B of the specified size. Used by ?bdsdc.
?lasd2	s,d	Merges the two sets of singular values together into a single sorted set. Used by ?bdsdc.
?lasd3	s,d	Finds all square roots of the roots of the secular equation, as defined by the values in D and Z, and then updates the singular vectors by matrix multiplication. Used by ?bdsdc.
?lasd4	s,d	Computes the square root of the i-th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix. Used by ?bdsdc.
?lasd5	s,d	Computes the square root of the i-th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix.Used by ?bdsdc.
?lasd6	s,d	Computes the SVD of an updated upper bidiagonal matrix obtained by merging two smaller ones by appending a row. Used by ?bdsdc.
?lasd7	s,d	Merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. Used by ?bdsdc.
?lasd8	s,d	Finds the square roots of the roots of the secular equation, and stores, for each element in D, the distance to its two nearest poles. Used by ?bdsdc.
?lasd9	s,d	Finds the square roots of the roots of the secular equation, and stores, for each element in D, the distance to its two nearest poles. Used by ?bdsdc.
?lasda	s,d	Computes the singular value decomposition (SVD) of a real upper bidiagonal matrix with diagonal *d* and off-diagonal *e*. Used by ?bdsdc.
?lasdq	s,d	Computes the SVD of a real bidiagonal matrix with diagonal *d* and off-diagonal *e*. Used by ?bdsdc.
?lasdt	s,d	Creates a tree of subproblems for bidiagonal divide and conquer. Used by ?bdsdc.
?laset	s,d,c,z	Initializes the off-diagonal elements and the diagonal elements of a matrix to given values.

Table 5-1 LAPACK Auxiliary Routines (continued)

Routine Name	Data Types	Description
?lasq1	s,d	Computes the singular values of a real square bidiagonal matrix. Used by ?bdsqr.
?lasq2	s,d	Computes all the eigenvalues of the symmetric positive definite tridiagonal matrix associated with the *qd* Array *z* to high relative accuracy. Used by ?bdsqr and ?stegr.
?lasq3	s,d	Checks for deflation, computes a shift and calls *dqds*. Used by ?bdsqr.
?lasq4	s,d	Computes an approximation to the smallest eigenvalue using values of d from the previous transform. Used by ?bdsqr.
?lasq5	s,d	Computes one *dqds* transform in ping-pong form. Used by ?bdsqr and ?stegr.
?lasq6	s,d	Computes one *dqds* transform in ping-pong form. Used by ?bdsqr and ?stegr.
?lasr	s,d,c,z	Applies a sequence of plane rotations to a general rectangular matrix.
?lasrt	s,d	Sorts numbers in increasing or decreasing order.
?lassq	s,d,c,z	Updates a sum of squares represented in scaled form.
?lasv2	s,d	Computes the singular value decomposition of a 2-by-2 triangular matrix.
?laswp	s,d,c,z	Performs a series of row interchanges on a general rectangular matrix.
?lasy2	s,d	Solves the Sylvester matrix equation where the matrices are of order 1 or 2.
?lasyf	s,d,c,z	Computes a partial factorization of a real/complex symmetric matrix, using the diagonal pivoting method.
?lahef	c,z	Computes a partial factorization of a complex Hermitian indefinite matrix, using the diagonal pivoting method.
?latbs	s,d,c,z	Solves a triangular banded system of equations.
?latdf	s,d,c,z	Uses the LU factorization of the n-by-n matrix computed by ?getc2 and computes a contribution to the reciprocal Dif-estimate.
?latps	s,d,c,z	Solves a triangular system of equations with the matrix held in packed storage.

Table 5-1 LAPACK Auxiliary Routines (continued)

Routine Name	Data Types	Description
?latrd	s,d,c,z	Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.
?latrs	s,d,c,z	Solves a triangular system of equations with the scale factor set to prevent overflow.
?latrz	s,d,c,z	Factors an upper trapezoidal matrix by means of orthogonal/unitary transformations.
?lauu2	s,d,c,z	Computes the product UU^H or L^HL, where U and L are upper or lower triangular matrices (unblocked algorithm).
?lauum	s,d,c,z	Computes the product UU^H or L^HL, where U and L are upper or lower triangular matrices.
?org2l/?ung2l	s,d/c,z	Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by ?geqlf (unblocked algorithm).
?org2r/?ung2r	s,d/c,z	Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by ?geqrf (unblocked algorithm).
?orgl2/?ungl2	s,d/c,z	Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by ?gelqf (unblocked algorithm).
?orgr2/?ungr2	s,d/c,z	Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by ?gerqf (unblocked algorithm).
?orm2l/?unm2l	s,d/c,z	Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by ?geqlf (unblocked algorithm).
?orm2r/?unm2r	s,d/c,z	Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by ?geqrf (unblocked algorithm).
?orml2/?unml2	s,d/c,z	Multiplies a general matrix by the orthogonal/unitary matrix from a LQ factorization determined by ?gelqf (unblocked algorithm).
?ormr2/?unmr2	s,d/c,z	Multiplies a general matrix by the orthogonal/unitary matrix from a RQ factorization determined by ?gerqf (unblocked algorithm).
?ormr3/?unmr3	s,d/c,z	Multiplies a general matrix by the orthogonal/unitary matrix from a RZ factorization determined by ?tzrzf (unblocked algorithm).
?pbtf2	s,d,c,z	Computes the Cholesky factorization of a symmetric/ Hermitian positive definite band matrix (unblocked algorithm).
?potf2	s,d,c,z	Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (unblocked algorithm).
?ptts2	s,d,c,z	Solves a tridiagonal system of the form $AX=B$ using the $L\ D\ L^H$ factorization computed by ?pttrf.

Table 5-1 LAPACK Auxiliary Routines (continued)

Routine Name	Data Types	Description
?rscl	s,d,cs, zd	Multiplies a vector by the reciprocal of a real scalar.
?sygs2/?hegs2	s,d/c,z	Reduces a symmetric/Hermitian definite generalized eigenproblem to standard form, using the factorization results obtained from ?potrf (unblocked algorithm).
?sytd2/?hetd2	s,d/c,z	Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (unblocked algorithm).
?sytf2	s,d,c,z	Computes the factorization of a real/complex symmetric indefinite matrix, using the diagonal pivoting method (unblocked algorithm).
?hetf2	c,z	Computes the factorization of a complex Hermitian matrix, using the diagonal pivoting method (unblocked algorithm).
?tgex2	s,d,c,z	Swaps adjacent diagonal blocks in an upper (quasi) triangular matrix pair by an orthogonal/unitary equivalence transformation.
?tgsy2	s,d,c,z	Solves the generalized Sylvester equation (unblocked algorithm).
?trti2	s,d,c,z	Computes the inverse of a triangular matrix (unblocked algorithm).

?lacgv

Conjugates a complex vector.

Syntax

```
call clacgv (n, x, incx)
call zlacgv (n, x, incx)
```

Description

This routine conjugates a complex vector *x* of length *n* and increment *incx* (see "Vector Arguments in BLAS" in Appendix B).

Input Parameters

n INTEGER. The length of the vector *x* ($n \ge 0$).

x COMPLEX for clacgv
COMPLEX*16 for zlacgv.
Array, dimension (1+(*n*-1)* |*incx*|).
Contains the vector of length *n* to be conjugated.

incx INTEGER. The spacing between successive elements of *x*.

Output Parameters

x On exit, overwritten with conjg(*x*).

?lacrm

Multiplies a complex matrix by a square real matrix.

Syntax

```
call clacrm (m, n, a, lda, b, ldb, c, ldc, rwork)
call zlacrm (m, n, a, lda, b, ldb, c, ldc, rwork)
```

Description

This routine performs a simple matrix-matrix multiplication of the form

$$C = A * B,$$

where A is m-by-n and complex, B is n-by-n and real, C is m-by-n and complex.

Input Parameters

m INTEGER. The number of rows of the matrix A and of the matrix C ($m \ge 0$).

n INTEGER. The number of columns and rows of the matrix B and the number of columns of the matrix C
($n \ge 0$).

a COMPLEX for clacrm
COMPLEX*16 for zlacrm

Array, DIMENSION (*lda*, *n*). Contains the m-by-n matrix A.

lda INTEGER. The leading dimension of the array *a*,
$lda \ge \max(1, m)$.

b `REAL` for `clacrm`
`DOUBLE PRECISION` for `zlacrm`

Array, `DIMENSION (`*ldb, n*`)`. Contains the *n*-by-*n* matrix *B*.

ldb `INTEGER`. The leading dimension of the array *b*, *ldb* ≥ `max(1,` *n*`)`.

ldc `INTEGER`. The leading dimension of the output array *c*, *ldc* ≥ `max(1,` *n*`)`.

rwork `REAL` for `clacrm`
`DOUBLE PRECISION` for `zlacrm`

Workspace array, `DIMENSION (2*`*m*`*`*n*`)`.

Output Parameters

c `COMPLEX` for `clacrm`
`COMPLEX*16` for `zlacrm`

Array, `DIMENSION (`*ldc, n*`)`. Contains the *m*-by-*n* matrix *C*.

?lacrt

Performs a linear transformation of a pair of complex vectors.

Syntax

```
call clacrt (n, cx, incx, cy, incy, c, s)
call zlacrt (n, cx, incx, cy, incy, c, s)
```

Description

This routine performs the following transformation

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} \Rightarrow \begin{pmatrix} x \\ y \end{pmatrix},$$

where *c*, *s* are complex scalars and *x*, *y* are complex vectors.

Input Parameters

n	`INTEGER`. The number of elements in the vectors *cx* and *cy* ($n \ge 0$).
cx, cy	`COMPLEX` for `clacrt` `COMPLEX*16` for `zlacrt` Arrays, dimension (*n*). Contain input vectors *x* and *y*, respectively.
incx	`INTEGER`. The increment between successive elements of *cx*.
incy	`INTEGER`. The increment between successive elements of *cy*.
c, s	`COMPLEX` for `clacrt` `COMPLEX*16` for `zlacrt` Complex scalars that define the transform matrix

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}$$

Output Parameters

cx	On exit, overwritten with $c*x + s*y$.
cy	On exit, overwritten with $-s*x + c*y$.

?laesy

Computes the eigenvalues and eigenvectors of a 2-by-2 complex symmetric matrix, and checks that the norm of the matrix of eigenvectors is larger than a threshold value.

Syntax

```
call claesy (a, b, c, rt1, rt2, evscal, cs1, sn1)
call zlaesy (a, b, c, rt1, rt2, evscal, cs1, sn1)
```

Description

This routine performs the eigendecomposition of a 2-by-2 symmetric matrix

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix},$$

provided the norm of the matrix of eigenvectors is larger than some threshold value.

rt1 is the eigenvalue of larger absolute value, and *rt2* of smaller absolute value. If the eigenvectors are computed, then on return (*cs1*, *sn1*) is the unit eigenvector for *rt1*, hence

$$\begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} \cdot \begin{bmatrix} a & b \\ b & c \end{bmatrix} \cdot \begin{bmatrix} cs1 & -sn1 \\ sn1 & cs1 \end{bmatrix} = \begin{bmatrix} rt1 & 0 \\ 0 & rt2 \end{bmatrix}$$

Input Parameters

a, b, c	`COMPLEX` for `claesy` `COMPLEX*16` for `zlaesy` Elements of the input matrix.

Output Parameters

rt1, rt2	`COMPLEX` for `claesy` `COMPLEX*16` for `zlaesy` Eigenvalues of larger and smaller modulus, respectively.
evscal	`COMPLEX` for `claesy` `COMPLEX*16` for `zlaesy` The complex value by which the eigenvector matrix was scaled to make it orthonormal. If *evscal* is zero, the eigenvectors were not computed. This means one of two things: the 2-by-2 matrix could not be diagonalized, or the norm of the matrix of eigenvectors before scaling was larger than the threshold value `thresh` (set to 0.1E0).
cs1, sn1	`COMPLEX` for `claesy` `COMPLEX*16` for `zlaesy` If *evscal* is not zero, then (*cs1*, *sn1*) is the unit right eigenvector for *rt1*.

?rot

Applies a plane rotation with real cosine and complex sine to a pair of complex vectors.

Syntax

```
call crot (n, cx, incx, cy, incy, c, s)
call zrot (n, cx, incx, cy, incy, c, s)
```

Description

This routine applies a plane rotation, where the cosine (*c*) is real and the sine (*s*) is complex, and the vectors *cx* and *cy* are complex. This routine has its real equivalents in BLAS (see `?rot` in Chapter 2).

Input Parameters

n	`INTEGER`. The number of elements in the vectors *cx* and *cy*.
cx, cy	`COMPLEX` for `crot` `COMPLEX*16` for `zrot` Arrays of dimension (*n*), contain input vectors *x* and *y*, respectively.
incx	`INTEGER`. The increment between successive elements of *cx*.
incy	`INTEGER`. The increment between successive elements of *cy*.
c	`REAL` for `crot` `DOUBLE PRECISION` for `zrot`
s	`COMPLEX` for `crot` `COMPLEX*16` for `zrot` Values that define a rotation

$$\begin{bmatrix} c & s \\ -\mathrm{conjg}(s) & c \end{bmatrix}$$

where $c*c + s*\mathrm{conjg}(s) = 1.0$.

Output Parameters

cx On exit, overwritten with `c*x + s*y`.

cy On exit, overwritten with `-conjg(s)*x + c*y`.

?spmv

Computes a matrix-vector product for complex vectors using a complex symmetric packed matrix.

Syntax

```
call cspmv ( uplo, n, alpha, ap, x, incx, beta, y, incy )
call zspmv ( uplo, n, alpha, ap, x, incx, beta, y, incy )
```

Description

These routines perform a matrix-vector operation defined as

`y := alpha*a*x + beta*y,`

where:

alpha and *beta* are complex scalars,

x and *y* are *n*-element complex vectors

a is an *n*-by-*n* complex symmetric matrix, supplied in packed form.

These routines have their real equivalents in BLAS (see `?spmv` in Chapter 2).

Input Parameters

uplo `CHARACTER*1`. Specifies whether the upper or lower triangular part of the matrix *a* is supplied in the packed array *ap*, as follows:

If *uplo* = `'U'` or `'u'`, the upper triangular part of the matrix *a* is supplied in the array *ap*.
If *uplo* = `'L'` or `'l'`, the lower triangular part of the matrix *a* is supplied in the array *ap*.

n `INTEGER`. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

alpha, beta — COMPLEX for cspmv
COMPLEX*16 for zspmv

Specify complex scalars *alpha* and *beta*. When *beta* is supplied as zero, then *y* need not be set on input.

ap — COMPLEX for cspmv
COMPLEX*16 for zspmv

Array, DIMENSION at least ((*n**(*n* + 1))/2). Before entry, with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(1, 2) and *a*(2, 2) respectively, and so on. Before entry, with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(2, 1) and *a*(3, 1) respectively, and so on.

x — COMPLEX for cspmv
COMPLEX*16 for zspmv

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array x must contain the *n*-element vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

y — COMPLEX for cspmv
COMPLEX*16 for zspmv

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy — INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y — Overwritten by the updated vector *y*.

?spr

Performs the symmetrical rank-1 update of a complex symmetric packed matrix.

Syntax

```
call cspr( uplo, n, alpha, x, incx, ap )
call zspr( uplo, n, alpha, x, incx, ap )
```

Description

The `?spr` routines perform a matrix-vector operation defined as

a:= *alpha***x**conjg(*x*') + *a*,

where:

alpha is a complex scalar

x is an *n*-element complex vector

a is an *n*-by-*n* complex symmetric matrix, supplied in packed form.

These routines have their real equivalents in BLAS (see `?spr` in Chapter 2).

Input Parameters

uplo	`CHARACTER*1`. Specifies whether the upper or lower triangular part of the matrix *a* is supplied in the packed array *ap*, as follows: If *uplo* = `'U'` or `'u'`, the upper triangular part of the matrix *a* is supplied in the array *ap* . If *uplo* = `'L'` or `'l'`, the lower triangular part of the matrix *a* is supplied in the array *ap* .
n	`INTEGER`. Specifies the order of the matrix *a*. The value of *n* must be at least zero.
alpha	`COMPLEX` for `cspr` `COMPLEX*16` for `zspr` Specifies the scalar *alpha*.

x — COMPLEX for cspr
COMPLEX*16 for zspr

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

ap — COMPLEX for cspr
COMPLEX*16 for zspr

Array, DIMENSION at least ((*n**(*n* + 1))/2). Before entry, with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a*(1, 2) and *a*(2,2) respectively, and so on.

Before entry, with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2)and *ap*(3)contain *a*(2,1) and *a*(3,1) respectively, and so on.

Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.

Output Parameters

ap — With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.

?symv

Computes a matrix-vector product for a complex symmetric matrix.

Syntax

```
call csymv ( uplo, n, alpha, a, lda, x, incx, beta, y, incy )
call zsymv ( uplo, n, alpha, a, lda, x, incx, beta, y, incy )
```

Description

These routines perform the matrix-vector operation defined as

*y := alpha*a*x + beta*y,*

where:

alpha and *beta* are complex scalars

x and *y* are *n*-element complex vectors

a is an *n*-by-*n* symmetric complex matrix.

These routines have their real equivalents in BLAS (see ?symv in Chapter 2).

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is to be referenced, as follows: If *uplo* = 'U' or 'u', the upper triangular part of the array *a* is to be referenced . If *uplo* = 'L' or 'l', the lower triangular part of the array *a* is to be referenced.
n	INTEGER. Specifies the order of the matrix *a*. The value of *n* must be at least zero.
alpha, beta	COMPLEX for csymv COMPLEX*16 for zsymv Specify the scalars *alpha* and *beta*. When *beta* is supplied as zero, then *y* need not be set on input.
a	COMPLEX for csymv COMPLEX*16 for zsymv Array, DIMENSION (*lda, n*). Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.
lda	INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1,*n*).

x — `COMPLEX` for `csymv`
`COMPLEX*16` for `zsymv`

Array, `DIMENSION` at least `(1 + (`*n* `- 1)*abs(`*incx*`))`. Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx — `INTEGER`. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

y — `COMPLEX` for `csymv`
`COMPLEX*16` for `zsymv`

Array, `DIMENSION` at least `(1 + (`*n* `- 1)*abs(`*incy*`))`. Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy — `INTEGER`. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y — Overwritten by the updated vector *y*.

?syr

Performs the symmetric rank-1 update of a complex symmetric matrix.

Syntax

```
call csyr( uplo, n, alpha, x, incx, a, lda )
call zsyr( uplo, n, alpha, x, incx, a, lda )
```

Description

These routines perform the symmetric rank 1 operation defined as

*a := alpha*x*x' + a,*

where:

alpha is a complex scalar

x is an *n*-element complex vector

a is an *n*-by-*n* complex symmetric matrix.

These routines have their real equivalents in BLAS (see ?syr in Chapter 2).

Input Parameters

uplo — CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is to be referenced, as follows:

If *uplo* = 'U' or 'u', the upper triangular part of the array *a* is to be referenced .
If *uplo* = 'L' or 'l', the lower triangular part of the array *a* is to be referenced.

n — INTEGER. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

alpha — COMPLEX for csyr
COMPLEX*16 for zsyr

Specifies the scalar *alpha*.

x — COMPLEX for csyr
COMPLEX*16 for zsyr

Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector x.

incx — INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

a — COMPLEX for csyr
COMPLEX*16 for zsyr

Array, DIMENSION (*lda, n*). Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced.

Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.

lda — INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1,*n*).

Output Parameters

a	With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix. With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.

i?max1

Finds the index of the vector element whose real part has maximum absolute value.

Syntax

```
index = icmax1 ( n, cx, incx )
index = izmax1 ( n, cx, incx )
```

Description

Given a complex vector *cx*, the `i?max1` functions return the index of the vector element whose real part has maximum absolute value. These functions are based on the BLAS functions `icamax/izamax`, but using the absolute value of the real part. They are designed for use with `clacon/zlacon`.

Input Parameters

n	`INTEGER`. Specifies the number of elements in the vector *cx*.
cx	`COMPLEX` for `icmax1` `COMPLEX*16` for `izmax1` Array, `DIMENSION` at least `(1+(n-1)*abs(incx))`. Contains the input vector.
incx	`INTEGER`. Specifies the spacing between successive elements of *cx*.

Output Parameters

index	`INTEGER`. Contains the index of the vector element whose real part has maximum absolute value.

?sum1

Forms the 1-norm of the complex vector using the true absolute value.

Syntax

res = scsum1 (*n*, *cx*, *incx*)

res = dzsum1 (*n*, *cx*, *incx*)

Description

Given a complex vector *cx*, scsum1/dzsum1 functions take the sum of the absolute values of vector elements and return a single/double precision result, respectively. These functions are based on scasum/dzasum from Level 1 BLAS, but use the true absolute value and were designed for use with clacon/zlacon.

Input Parameters

n INTEGER. Specifies the number of elements in the vector *cx*.

cx COMPLEX for scsum1
COMPLEX*16 for dzsum1

Array, DIMENSION at least (1+(*n*-1)*abs(*incx*)).

Contains the input vector whose elements will be summed.

incx INTEGER. Specifies the spacing between successive elements of *cx* (*incx* > 0).

Output Parameters

res REAL for scsum1
DOUBLE PRECISION for dzsum1

Contains the sum of absolute values.

?gbtf2

Computes the LU factorization of a general band matrix using the unblocked version of the algorithm.

Syntax

```
call sgbtf2 ( m, n, kl, ku, ab, ldab, ipiv, info )
call dgbtf2 ( m, n, kl, ku, ab, ldab, ipiv, info )
call cgbtf2 ( m, n, kl, ku, ab, ldab, ipiv, info )
call zgbtf2 ( m, n, kl, ku, ab, ldab, ipiv, info )
```

Description

The routine forms the *LU* factorization of a general real/complex *m* by *n* band matrix *A* with *kl* sub-diagonals and *ku* super-diagonals. The routine uses partial pivoting with row interchanges and implements the unblocked version of the algorithm, calling Level 2 BLAS. See also `?gbtrf`.

Input Parameters

m	`INTEGER`. The number of rows of the matrix *A* ($m \geq 0$).
n	`INTEGER`. The number of columns in *A* ($n \geq 0$).
kl	`INTEGER`. The number of sub-diagonals within the band of *A* ($kl \geq 0$).
ku	`INTEGER`. The number of super-diagonals within the band of *A* ($ku \geq 0$).
ab	`REAL` for `sgbtf2` `DOUBLE PRECISION` for `dgbtf2` `COMPLEX` for `cgbtf2` `COMPLEX*16` for `zgbtf2`. Array, `DIMENSION` (*ldab*, *). The array *ab* contains the matrix *A* in band storage (see Matrix Arguments). The second dimension of *ab* must be at least max(1, *n*).
ldab	`INTEGER`. The first dimension of the array *ab*. ($ldab \geq 2kl + ku + 1$)

Output Parameters

ab	Overwritten by details of the factorization. The diagonal and *kl* + *ku* super-diagonals of *U* are stored in the first 1 + *kl* + *ku* rows of *ab*. The multipliers used during the factorization are stored in the next *kl* rows.

ipiv `INTEGER`.
Array, `DIMENSION` at least max(1,min(m,n)).
The pivot indices: row i was interchanged with row `ipiv(i)`.

info `INTEGER`. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, u_{ii} is 0. The factorization has been completed, but U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

?gebd2

Reduces a general matrix to bidiagonal form using an unblocked algorithm.

Syntax

```
call sgebd2 ( m, n, a, lda, d, e, tauq, taup, work, info )
call dgebd2 ( m, n, a, lda, d, e, tauq, taup, work, info )
call cgebd2 ( m, n, a, lda, d, e, tauq, taup, work, info )
call zgebd2 ( m, n, a, lda, d, e, tauq, taup, work, info )
```

Description

The routine reduces a general m-by-n matrix A to upper or lower bidiagonal form B by an orthogonal (unitary) transformation: $Q' A P = B$

If $m \geq n$, B is upper bidiagonal; if $m < n$, B is lower bidiagonal.

The routine does not form the matrices Q and P explicitly, but represents them as products of elementary reflectors. If $m \geq n$,

$Q = H(1)H(2)...H(n)$ and $P = G(1)G(2)...G(n-1)$

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

$H(i) = I - tauq*v*v'$ and $G(i) = I - taup*u*u'$

where *tauq* and *taup* are scalars (real for `sgebd2/dgebd2`, complex for `cgebd2/zgebd2`), and *v* and *u* are vectors (real for `sgebd2/dgebd2`, complex for `cgebd2/zgebd2`).

Input Parameters

m — `INTEGER`. The number of rows in the matrix *A* ($m \ge 0$).

n — `INTEGER`. The number of columns in *A* ($n \ge 0$).

a, work — `REAL` for `sgebd2`
`DOUBLE PRECISION` for `dgebd2`
`COMPLEX` for `cgebd2`
`COMPLEX*16` for `zgebd2`.

Arrays:
a(*lda*,*) contains the *m*-by-*n* general matrix *A* to be reduced. The second dimension of *a* must be at least max(1, *n*).

work(*) is a workspace array, the dimension of *work* must be at least max(1, *m*, *n*).

lda — `INTEGER`. The first dimension of *a*; at least max(1, *m*).

Output Parameters

a — If $m \ge n$, the diagonal and first super-diagonal of *a* are overwritten with the upper bidiagonal matrix *B*. Elements below the diagonal, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and elements above the first superdiagonal, with the array *taup*, represent the orthogonal/unitary matrix *P* as a product of elementary reflectors.

If $m < n$, the diagonal and first sub-diagonal of *a* are overwritten by the lower bidiagonal matrix *B*. Elements below the first subdiagonal, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and elements above the diagonal, with the array *taup*, represent the orthogonal/unitary matrix *P* as a product of elementary reflectors.

d — `REAL` for single-precision flavors
`DOUBLE PRECISION` for double-precision flavors. Array, `DIMENSION` at least max(1, min(*m*, *n*)).
Contains the diagonal elements of the bidiagonal matrix *B*: *d*(i) = *a*(i, i).

e — `REAL` for single-precision flavors
`DOUBLE PRECISION` for double-precision flavors. Array, `DIMENSION` at least max(1, min(*m*, *n*) – 1).
Contains the off-diagonal elements of the bidiagonal matrix *B*:

If $m \geq n$, $e(i) = a(i, i+1)$ for i = 1,2,..., n-1;
If $m < n$, $e(i) = a(i+1, i)$ for i = 1,2,..., m-1.

tauq, taup — `REAL` for `sgebd2`
`DOUBLE PRECISION` for `dgebd2`
`COMPLEX` for `cgebd2`
`COMPLEX*16` for `zgebd2`.
Arrays, `DIMENSION` at least max (1, min(m, n)).
Contain scalar factors of the elementary reflectors which represent orthogonal/unitary matrices Q and P, respectively.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

?gehd2

Reduces a general square matrix to upper Hessenberg form using an unblocked algorithm.

Syntax

```
call sgehd2 ( n, ilo, ihi, a, lda, tau, work, info )
call dgehd2 ( n, ilo, ihi, a, lda, tau, work, info )
call cgehd2 ( n, ilo, ihi, a, lda, tau, work, info )
call zgehd2 ( n, ilo, ihi, a, lda, tau, work, info )
```

Description

The routine reduces a real/complex general matrix A to upper Hessenberg form H by an orthogonal or unitary similarity transformation $Q' A Q = H$.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of *elementary reflectors*.

Input Parameters

n — `INTEGER`. The order of the matrix A ($n \geq 0$).

ilo, ihi	INTEGER. It is assumed that *A* is already upper triangular in rows and columns 1:*ilo* -1 and *ihi*+1:*n* . If *A* has been output by ?gebal, then *ilo* and *ihi* must contain the values returned by that routine. Otherwise they should be set to *ilo* = 1 and *ihi* = *n*. Constraint: $1 \le ilo \le ihi \le \max(1, n)$.
a, work	REAL for sgehd2 DOUBLE PRECISION for dgehd2 COMPLEX for cgehd2 COMPLEX*16 for zgehd2. Arrays: *a* (*lda*,*) contains the *n*-by-*n* matrix *A* to be reduced. The second dimension of *a* must be at least max(1, *n*). *work* (*n*) is a workspace array.
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).

Output Parameters

a	On exit, the upper triangle and the first subdiagonal of *A* are overwritten with the upper Hessenberg matrix *H* and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors. See *Application Notes* below.
tau	REAL for sgehd2 DOUBLE PRECISION for dgehd2 COMPLEX for cgehd2 COMPLEX*16 for zgehd2. Array, DIMENSION at least max (1, *n*-1). Contains the scalar factors of elementary reflectors. See *Application Notes* below.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The matrix *Q* is represented as a product of (*ihi* -*ilo*) elementary reflectors

$Q = H(ilo)\, H(ilo+1) \ldots H(ihi-1)$

Each *H*(i) has the form

$H(\mathrm{i}) = I - tau * v * v'$

where *tau* is a real/complex scalar, and v is a real/complex vector with $v(1{:}\mathrm{i}) = 0$, $v(\mathrm{i}+1) = 1$ and $v(ihi+1{:}n) = 0$.

On exit, $v(\mathrm{i}+2{:}ihi)$ is stored in $a(\mathrm{i}+2{:}ihi, \mathrm{i})$ and *tau* in `tau`(i).

The contents of *a* are illustrated by the following example, with $n = 7$, `ilo` = 2 and `ihi` = 6:

on entry | on exit

$$
\begin{bmatrix}
a & a & a & a & a & a & a \\
 & a & a & a & a & a & a \\
 & a & a & a & a & a & a \\
 & a & a & a & a & a & a \\
 & a & a & a & a & a & a \\
 & a & a & a & a & a & a \\
 & & & & & & a
\end{bmatrix}
\qquad
\begin{bmatrix}
a & a & h & h & h & h & a \\
 & a & h & h & h & h & a \\
 & h & h & h & h & h & h \\
 & v_2 & h & h & h & h & h \\
 & v_2 & v_3 & h & h & h & h \\
 & v_2 & v_3 & v_4 & h & h & h \\
 & & & & & & a
\end{bmatrix}
$$

where a denotes an element of the original matrix *A*, h denotes a modified element of the upper Hessenberg matrix *H*, and v_i denotes an element of the vector defining *H*(i).

?gelq2

Computes the LQ factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgelq2 ( m, n, a, lda, tau, work, info )
call dgelq2 ( m, n, a, lda, tau, work, info )
call cgelq2 ( m, n, a, lda, tau, work, info )
call zgelq2 ( m, n, a, lda, tau, work, info )
```

Description

The routine computes an LQ factorization of a real/complex m by n matrix A as $A = L Q$.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of min(m, n) *elementary reflectors* :

$Q = H(\mathrm{k}) \ldots H(2)\, H(1)$ (or $Q = H(\mathrm{k})' \ldots H(2)'\, H(1)'$ for complex flavors), where $\mathrm{k} = \min(m, n)$

Each H(i) has the form

$H(\mathrm{i}) = I - tau*v*v'$

where *tau* is a real/complex scalar stored in `tau`(i), and v is a real/complex vector with $v(1:\mathrm{i}-1) = 0$ and $v(\mathrm{i}) = 1$.

On exit, $v(\mathrm{i}+1:n)$ is stored in $a(\mathrm{i}, \mathrm{i}+1:n)$.

Input Parameters

m	`INTEGER`. The number of rows in the matrix A ($m \geq 0$).
n	`INTEGER`. The number of columns in A ($n \geq 0$).
a, work	`REAL` for `sgelq2` `DOUBLE PRECISION` for `dgelq2` `COMPLEX` for `cgelq2` `COMPLEX*16` for `zgelq2`. Arrays: `a(lda,*)` contains the m-by-n matrix A. The second dimension of `a` must be at least max(1, n).

`work(m)` is a workspace array.

lda — `INTEGER`. The first dimension of *a*; at least max(1, *m*).

Output Parameters

a — Overwritten by the factorization data as follows:

on exit, the elements on and below the diagonal of the array *a* contain the *m*-by-min(*n*,*m*) lower trapezoidal matrix *L* (*L* is lower triangular if $n \ge m$); the elements above the diagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of min(*n*,*m*) elementary reflectors.

tau — `REAL` for `sgelq2`
`DOUBLE PRECISION` for `dgelq2`
`COMPLEX` for `cgelq2`
`COMPLEX*16` for `zgelq2`.
Array, `DIMENSION` at least max(1, min(*m*, *n*)).
Contains scalar factors of the elementary reflectors.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.

?geql2

Computes the QL factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgeql2 ( m, n, a, lda, tau, work, info )
call dgeql2 ( m, n, a, lda, tau, work, info )
call cgeql2 ( m, n, a, lda, tau, work, info )
call zgeql2 ( m, n, a, lda, tau, work, info )
```

Description

The routine computes a *QL* factorization of a real/complex *m* by *n* matrix *A as* $A = Q L$.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of min(*m*, *n*) *elementary reflectors* :

$Q = H(k) \ldots H(2)\, H(1)$, where k = min(*m*, *n*)

Each $H(i)$ has the form

$H(i) = I - tau*v*v'$

where *tau* is a real/complex scalar stored in `tau`(i), and v is a real/complex vector with $v(m\text{-}k\text{+}i\text{+}1\text{:}m) = 0$ and $v(m\text{-}k\text{+}i) = 1$.

On exit, $v(1\text{:}m\text{-}k\text{+}i\text{-}1)$ is stored in *a*(1:*m*-k+i-1, *n*-k+i).

Input Parameters

m	`INTEGER`. The number of rows in the matrix A ($m \geq 0$).
n	`INTEGER`. The number of columns in A ($n \geq 0$).
a, work	`REAL` for `sgeql2` `DOUBLE PRECISION` for `dgeql2` `COMPLEX` for `cgeql2` `COMPLEX*16` for `zgeql2`. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix A. The second dimension of *a* must be at least max(1, *n*). *work*(*m*) is a workspace array.
lda	`INTEGER`. The first dimension of *a*; at least max(1, *m*).

Output Parameters

a	Overwritten by the factorization data as follows: on exit, if $m \geq n$, the lower triangle of the subarray *a*(*m*-*n*+1:*m*, 1:*n*) contains the *n*-by-*n* lower triangular matrix L; if $m < n$, the elements on and below the (*n*-*m*)th superdiagonal contain the *m*-by-*n* lower trapezoidal matrix L; the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors.
tau	`REAL` for `sgeql2` `DOUBLE PRECISION` for `dgeql2` `COMPLEX` for `cgeql2`

COMPLEX*16 for zgeql2.
Array, DIMENSION at least max(1, min(*m*, *n*)).
Contains scalar factors of the elementary reflectors.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

?geqr2

Computes the QR factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgeqr2 ( m, n, a, lda, tau, work, info )
call dgeqr2 ( m, n, a, lda, tau, work, info )
call cgeqr2 ( m, n, a, lda, tau, work, info )
call zgeqr2 ( m, n, a, lda, tau, work, info )
```

Description

The routine computes a *QR* factorization of a real/complex *m* by *n* matrix *A as* $A = Q R$.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) *elementary reflectors* :

$Q = H(1)H(2) \dots H(k)$, where k = min(*m*, *n*)

Each *H*(i) has the form

$H(i) = I - tau*v*v'$

where *tau* is a real/complex scalar stored in *tau*(i), and *v* is a real/complex vector with *v*(1:i-1) = 0 and *v*(i) = 1.

On exit, *v*(i+1:*m*) is stored in *a*(i+1:*m*, i).

Input Parameters

m INTEGER. The number of rows in the matrix *A* ($m \geq 0$).

n	`INTEGER`. The number of columns in A ($n \geq 0$).
a, work	`REAL` for `sgeqr2` `DOUBLE PRECISION` for `dgeqr2` `COMPLEX` for `cgeqr2` `COMPLEX*16` for `zgeqr2`. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix A. The second dimension of *a* must be at least max(1, *n*). *work*(*n*) is a workspace array.
lda	`INTEGER`. The first dimension of *a*; at least max(1, *m*).

Output Parameters

a	Overwritten by the factorization data as follows: on exit, the elements on and above the diagonal of the array *a* contain the min(*n*,*m*)-by-*n* upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array *tau*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors.
tau	`REAL` for `sgeqr2` `DOUBLE PRECISION` for `dgeqr2` `COMPLEX` for `cgeqr2` `COMPLEX*16` for `zgeqr2`. Array, `DIMENSION` at least max(1, min(*m*, *n*)). Contains scalar factors of the elementary reflectors.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

?gerq2

Computes the RQ factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgerq2 ( m, n, a, lda, tau, work, info )
call dgerq2 ( m, n, a, lda, tau, work, info )
call cgerq2 ( m, n, a, lda, tau, work, info )
call zgerq2 ( m, n, a, lda, tau, work, info )
```

Description

The routine computes a RQ factorization of a real/complex m by n matrix A *as* $A = R\,Q$.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of min(m, n) *elementary reflectors* :

$Q = H(1)H(2)\ \dots\ H(\mathrm{k})$, where k = min(m, n)

Each H(i) has the form

$H(\mathrm{i}) = I - tau*v*v'$

where *tau* is a real/complex scalar stored in `tau`(i), and v is a real/complex vector with $v(n\text{-k+i+1}:n) = 0$ and $v(n\text{-k+i}) = 1$.

On exit, $v(1:n\text{-k+i-1})$ is stored in `a`(m-k+i, 1:n-k+i-1).

Input Parameters

m	`INTEGER`. The number of rows in the matrix A ($m \geq 0$).
n	`INTEGER`. The number of columns in A ($n \geq 0$).
a, work	`REAL` for `sgerq2` `DOUBLE PRECISION` for `dgerq2` `COMPLEX` for `cgerq2` `COMPLEX*16` for `zgerq2`. Arrays: `a(lda,*)` contains the m-by-n matrix A. The second dimension of `a` must be at least max(1, n).

`work(m)` is a workspace array.

lda — `INTEGER`. The first dimension of *a*; at least max(1, *m*).

Output Parameters

a — Overwritten by the factorization data as follows:

on exit, if $m \le n$, the upper triangle of the subarray $a(1:m, n-m+1:n)$ contains the m-by-m upper triangular matrix *R*;
if $m > n$, the elements on and above the $(m-n)$th subdiagonal contain the m-by-n upper trapezoidal matrix *R;* the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.

tau — `REAL` for `sgerq2`
`DOUBLE PRECISION` for `dgerq2`
`COMPLEX` for `cgerq2`
`COMPLEX*16` for `zgerq2`.
Array, `DIMENSION` at least max(1, min(*m*, *n*)).
Contains scalar factors of the elementary reflectors.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.

?gesc2

Solves a system of linear equations using the LU factorization with complete pivoting computed by `?getc2`.

Syntax

```
call sgesc2 ( n, a, lda, rhs, ipiv, jpiv, scale )
call dgesc2 ( n, a, lda, rhs, ipiv, jpiv, scale )
call cgesc2 ( n, a, lda, rhs, ipiv, jpiv, scale )
call zgesc2 ( n, a, lda, rhs, ipiv, jpiv, scale )
```

Description

This routine solves a system of linear equations

$$AX = scale * RHS$$

with a general n-by-n matrix A using the *LU* factorization with complete pivoting computed by `?getc2`.

Input Parameters

n — `INTEGER`. The order of the matrix *A*.

a, rhs — `REAL` for `sgesc2`
`DOUBLE PRECISION` for `dgesc2`
`COMPLEX` for `cgesc2`
`COMPLEX*16` for `zgesc2`.
Arrays:
`a(lda,*)` contains the *LU* part of the factorization of the *n*-by-*n* matrix *A* computed by `?getc2`:
$A = P L U Q$.
The second dimension of *a* must be at least max(1, *n*);

`rhs(n)` contains on entry the right hand side vector for the system of equations.

lda — `INTEGER`. The first dimension of *a*; at least max(1, *n*).

ipiv — `INTEGER`.
Array, `DIMENSION` at least max(1,*n*).
The pivot indices: for $1 \le i \le n$, row i of the matrix has been interchanged with row `ipiv(i)`.

jpiv — `INTEGER`.
Array, `DIMENSION` at least max(1,*n*).
The pivot indices: for $1 \le j \le n$, column j of the matrix has been interchanged with column `jpiv(j)`.

Output Parameters

rhs — On exit, overwritten with the solution vector *X*.

scale — `REAL` for `sgesc2/cgesc2`
`DOUBLE PRECISION` for `dgesc2/zgesc2`
Contains the scale factor. *scale* is chosen in the range $0 \le scale \le 1$ to prevent overflow in the solution.

?getc2

Computes the LU factorization with complete pivoting of the general n-by-n matrix.

Syntax

```
call sgetc2 ( n, a, lda, ipiv, jpiv, info )
call dgetc2 ( n, a, lda, ipiv, jpiv, info )
call cgetc2 ( n, a, lda, ipiv, jpiv, info )
call zgetc2 ( n, a, lda, ipiv, jpiv, info )
```

Description

This routine computes an LU factorization with complete pivoting of the n-by-n matrix A. The factorization has the form $A = P * L * U * Q$, where P and Q are permutation matrices, L is lower triangular with unit diagonal elements and U is upper triangular.

Input Parameters

n — `INTEGER`. The order of the matrix A ($n \geq 0$).

a — `REAL` for `sgetc2`
`DOUBLE PRECISION` for `dgetc2`
`COMPLEX` for `cgetc2`
`COMPLEX*16` for `zgetc2`.
Array `a(lda,*)` contains the n-by-n matrix A to be factored.
The second dimension of *a* must be at least max(1, *n*);

lda — `INTEGER`. The first dimension of *a*; at least max(1, *n*).

Output Parameters

a — On exit, the factors L and U from the factorization $A = P*L*U*Q$; the unit diagonal elements of L are not stored. If U(k, k) appears to be less than `smin`, U(k, k) is given the value of `smin`, i.e., giving a nonsingular perturbed system.

ipiv — `INTEGER`.
Array, `DIMENSION` at least max(1,*n*).
The pivot indices: for $1 \leq i \leq n$, row `i` of the matrix has been interchanged with row `ipiv(i)`.

jpiv INTEGER.
Array, DIMENSION at least max(1,*n*).
The pivot indices: for $1 \le j \le n$, column j of the matrix has been interchanged with column *jpiv*(j).

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = $k > 0$, $U(k, k)$ is likely to produce overflow if we try to solve for *x* in $Ax = b$. So *U* is perturbed to avoid the overflow.

?getf2

Computes the LU factorization of a general m by n matrix using partial pivoting with row interchanges (unblocked algorithm).

Syntax

```
call sgetf2 ( m, n, a, lda, Ipiv, info )
call dgetf2 ( m, n, a, lda, Ipiv, info )
call cgetf2 ( m, n, a, lda, Ipiv, info )
call zgetf2 ( m, n, a, lda, Ipiv, info )
```

Description

The routine computes the *LU* factorization of a general *m*-by-*n* matrix *A using* partial pivoting with row interchanges. The factorization has the form

$$A = PLU,$$

where *P* is a permutation matrix, *L* is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and *U* is upper triangular (upper trapezoidal if $m < n$).

Input Parameters

m INTEGER. The number of rows in the matrix *A* ($m \ge 0$).

n INTEGER. The number of columns in *A* ($n \ge 0$).

a REAL for sgetf2
DOUBLE PRECISION for dgetf2
COMPLEX for cgetf2

	`COMPLEX*16` for `zgetf2`. Array, `DIMENSION` (*lda*, *). Contains the matrix A to be factored. The second dimension of *a* must be at least max(1, *n*).
lda	`INTEGER`. The first dimension of *a*; at least max(1, *m*).

Output Parameters

a	Overwritten by L and U. The unit diagonal elements of L are not stored.
ipiv	`INTEGER`. Array, `DIMENSION` at least max(1,min(*m*,*n*)). The pivot indices: for $1 \le i \le n$, row *i* was interchanged with row *ipiv*(*i*).
info	`INTEGER`. If *info*=0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i* > 0, u_{ii} is 0. The factorization has been completed, but U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

?gtts2

Solves a system of linear equations with a tridiagonal matrix using the LU factorization computed by `?gttrf`.

Syntax

```
call sgtts2 (itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb)
call dgtts2 (itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb)
call cgtts2 (itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb)
call zgtts2 (itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb)
```

Description

This routine solves for X one of the following systems of linear equations with multiple right hand sides:

$AX = B$ $A^TX = B$ or $A^HX = B$ (for complex matrices only),
with a tridiagonal matrix A using the LU factorization computed by `?gttrf`.

Input Parameters

itrans — INTEGER. Must be 0, 1, or 2.
Indicates the form of the equations being solved:
If *itrans* = 0, then $AX = B$ (no transpose).
If *itrans* = 1, then $A^T X = B$ (transpose).
If *itrans* = 2, then $A^H X = B$ (conjugate transpose).

n — INTEGER. The order of the matrix A ($n \geq 0$).

nrhs — INTEGER. The number of right-hand sides, i.e., the number of columns in B ($nrhs \geq 0$).

dl,d,du,du2,b — REAL for sgtts2
DOUBLE PRECISION for dgtts2
COMPLEX for cgtts2
COMPLEX*16 for zgtts2.
Arrays: *dl*(*n* - 1), *d*(*n*), *du*(*n* - 1), *du2*(*n* - 2), *b*(*ldb*, *nrhs*).
The array *dl* contains the (n - 1) multipliers that define the matrix L from the LU factorization of A.
The array *d* contains the *n* diagonal elements of the upper triangular matrix U from the LU factorization of A.
The array *du* contains the (n - 1) elements of the first super-diagonal of U.
The array *du2* contains the (n - 2) elements of the second super-diagonal of U.
The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations.

ldb — INTEGER. The leading dimension of *b*; must be *ldb* ≥ max(1, *n*).

ipiv — INTEGER.
Array, DIMENSION (*n*).
The pivot indices array, as returned by ?gttrf.

Output Parameters

b — Overwritten by the solution matrix X.

?labrd

Reduces the first nb rows and columns of a general matrix to a bidiagonal form.

Syntax

```
call slabrd ( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
call dlabrd ( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
call clabrd ( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
call zlabrd ( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
```

Description

The routine reduces the first *nb* rows and columns of a general *m*-by-*n* matrix A to upper or lower bidiagonal form by an orthogonal/unitary transformation $Q' A P$, and returns the matrices X and Y which are needed to apply the transformation to the unreduced part of A.

If $m \geq n$, A is reduced to upper bidiagonal form; if $m < n$, to lower bidiagonal form.

The matrices Q and P are represented as products of elementary reflectors: $Q = H(1)\ H(2) \ldots H(nb)$ and $P = G(1)\ G(2) \ldots G(nb)$

Each $H(\mathrm{i})$ and $G(\mathrm{i})$ has the form

$H(\mathrm{i}) = I - tauq * v * v'$ and $G(\mathrm{i}) = I - taup * u * u'$

where *tauq* and *taup* are scalars, and v and u are vectors.

The elements of the vectors v and u together form the *m*-by-*nb* matrix V and the *nb*-by-*n* matrix U' which are needed, with X and Y, to apply the transformation to the unreduced part of the matrix, using a block update of the form: $A := A - V*Y' - X*U'$.

This is an auxiliary routine called by `?gebrd`.

Input Parameters

m	`INTEGER`. The number of rows in the matrix A ($m \geq 0$).
n	`INTEGER`. The number of columns in A ($n \geq 0$).
nb	`INTEGER`. The number of leading rows and columns of A to be reduced.

a REAL for slabrd
DOUBLE PRECISION for dlabrd
COMPLEX for clabrd
COMPLEX*16 for zlabrd.

Array a(*lda*,*) contains the matrix *A* to be reduced.
The second dimension of a must be at least max(1, *n*).

lda INTEGER. The first dimension of a; at least max(1, *m*).

ldx INTEGER. The first dimension of the output array *x*; must beat least max(1, *m*).

ldy INTEGER. The first dimension of the output array *y*; must beat least max(1, *n*).

Output Parameters

a On exit, the first *nb* rows and columns of the matrix are overwritten; the rest of the array is unchanged.

If $m \geq n$, elements on and below the diagonal in the first *nb* columns, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors; and elements above the diagonal in the first *nb* rows, with the array *taup*, represent the orthogonal/unitary matrix *P* as a product of elementary reflectors.

If $m < n$, elements below the diagonal in the first *nb* columns, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and elements on and above the diagonal in the first *nb* rows, with the array *taup*, represent the orthogonal/unitary matrix *P* as a product of elementary reflectors.

d, e REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors. Arrays, DIMENSION (*nb*) each.
The array *d* contains the diagonal elements of the first *nb* rows and columns of the reduced matrix:
d(i) = *a*(i,i).
The array *e* contains the off-diagonal elements of the first *nb* rows and columns of the reduced matrix.

tauq, taup REAL for slabrd
DOUBLE PRECISION for dlabrd
COMPLEX for clabrd
COMPLEX*16 for zlabrd.

Arrays, `DIMENSION` (*nb*) each.
Contain scalar factors of the elementary reflectors which represent the orthogonal/unitary matrices *Q* and *P*, respectively.

x, *y*

`REAL` for `slabrd`
`DOUBLE PRECISION` for `dlabrd`
`COMPLEX` for `clabrd`
`COMPLEX*16` for `zlabrd`.

Arrays, dimension *x*(*ldx*, *nb*), *y*(*ldy*, *nb*).
The array *x* contains the *m*-by-*nb* matrix *X* required to update the unreduced part of *A*.

The array *y* contains the *n*-by-*nb* matrix *Y* required to update the unreduced part of *A*.

Application Notes

If $m \ge n$, then for the elementary reflectors *H*(i) and *G*(i),

$v(1:i-1) = 0$, $v(i) = 1$, and $v(i:m)$ is stored on exit in $a(i:m, i)$;
$u(1:i) = 0$, $u(i+1) = 1$, and $u(i+1:n)$ is stored on exit in $a(i, i+1:n)$;
tauq is stored in *tauq*(i) and *taup* in *taup*(i).

If $m < n$,

$v(1:i) = 0$, $v(i+1) = 1$, and $v(i+1:m)$ is stored on exit in $a(i+2:m, i)$;
$u(1:i-1) = 0$, $u(i) = 1$, and $u(i:n)$ is stored on exit in $a(i, i+1:n)$;
tauq is stored in *tauq*(i) and *taup* in *taup*(i).

The contents of *a* on exit are illustrated by the following examples with *nb* = 2:

$m = 6, n = 5$ ($m > n$)

$$\begin{bmatrix} 1 & 1 & u_1 & u_1 & u_1 \\ v_1 & 1 & 1 & u_2 & u_2 \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

$m = 5, n = 6$ ($m < n$)

$$\begin{bmatrix} 1 & u_1 & u_1 & u_1 & u_1 & u_1 \\ 1 & 1 & u_2 & u_2 & u_2 & u_2 \\ v_1 & 1 & a & a & a & a \\ v_1 & v_2 & a & a & a & a \\ v_1 & v_2 & a & a & a & a \end{bmatrix}$$

where a denotes an element of the original matrix which is unchanged, v_i denotes an element of the vector defining *H*(i), and u_i an element of the vector defining *G*(i).

?lacon

Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.

Syntax

```
call slacon ( n, v, x, isgn, est, kase )
call dlacon ( n, v, x, isgn, est, kase )
call clacon ( n, v, x, est, kase )
call zlacon ( n, v, x, est, kase )
```

Description

This routine estimates the 1-norm of a square, real/complex matrix *A*. Reverse communication is used for evaluating matrix-vector products.

Input Parameters

n — `INTEGER`. The order of the matrix *A* ($n \geq 1$).

v, *x* — `REAL` for `slacon`
`DOUBLE PRECISION` for `dlacon`
`COMPLEX` for `clacon`
`COMPLEX*16` for `zlacon`.

Arrays, `DIMENSION` (*n*) each.
v is a workspace array.
x is used as input after an intermediate return.

isgn — `INTEGER`. Workspace array, `DIMENSION` (*n*) , used with real flavors only.

kase — `INTEGER`.
On the initial call to `?lacon`, *kase* should be 0.

Output Parameters

est — `REAL` for `slacon/clacon`
`DOUBLE PRECISION` for `dlacon/zlacon`
An estimate (a lower bound) for norm(A).

kase — On an intermediate return, *kase* will be 1 or 2, indicating whether *x* should be overwritten by A **x* or A'**x*. On the final return from `?lacon`, *kase* will again be 0.

v — On the final return, *v* = A*w, where *est* = norm(*v*)/norm(w) (w is not returned).

x — On an intermediate return, *x* should be overwritten by
A **x* , if *kase* = 1,
A'**x* , if *kase* = 2,
(where for complex flavors A' is the conjugate transpose of A), and `?lacon` must be re-called with all the other parameters unchanged.

?lacpy

Copies all or part of one two-dimensional array to another.

Syntax

```
call slacpy ( uplo, m, n, a, lda, b, ldb )
call dlacpy ( uplo, m, n, a, lda, b, ldb )
call clacpy ( uplo, m, n, a, lda, b, ldb )
call zlacpy ( uplo, m, n, a, lda, b, ldb )
```

Description

This routine copies all or part of a two-dimensional matrix A to another matrix B.

Input Parameters

uplo — `CHARACTER*1`.
Specifies the part of the matrix A to be copied to B.

If *uplo* = 'U', the upper triangular part of *A* is copied.
If *uplo* = 'L', the lower triangular part of *A* is copied.
Otherwise, all of the matrix *A* is copied.

m — INTEGER. The number of rows in the matrix *A* ($m \ge 0$).

n — INTEGER. The number of columns in *A* ($n \ge 0$).

a — REAL for slacpy
DOUBLE PRECISION for dlacpy
COMPLEX for clacpy
COMPLEX*16 for zlacpy.
Array a(*lda*,*), contains the *m*-by-*n* matrix *A*.
The second dimension of a must be at least max(1,*n*).
If *uplo* = 'U', only the upper triangle or trapezoid is accessed; if *uplo* = 'L', only the lower triangle or trapezoid is accessed.

lda — INTEGER. The first dimension of a; $lda \ge \max(1, m)$.

ldb — INTEGER. The first dimension of the output array b; $ldb \ge \max(1, m)$.

Output Parameters

b — REAL for slacpy
DOUBLE PRECISION for dlacpy
COMPLEX for clacpy
COMPLEX*16 for zlacpy.
Array b(*ldb*,*), contains the *m*-by-*n* matrix *B*.
The second dimension of b must be at least max(1,*n*).
On exit, $B = A$ in the locations specified by *uplo* .

?ladiv

Performs complex division in real arithmetic, avoiding unnecessary overflow.

Syntax

```
call sladiv ( a, b, c, d, p, q )
call dladiv ( a, b, c, d, p, q )
res = cladiv ( x, y )
```

```
res = zladiv ( x, y )
```

Description

The routines `sladiv/dladiv` perform complex division in real arithmetic as

$$p + iq = \frac{a + ib}{c + id}$$

Complex functions `cladiv/zladiv` compute the result as

$$res = x/y ,$$

where x and y are complex. The computation of x / y will not overflow on an intermediary step unless the results overflows.

Input Parameters

a, b, c, d — `REAL` for `sladiv`
`DOUBLE PRECISION` for `dladiv`
The scalars a, b, c, and d in the above expression (for real flavors only).

x, y — `COMPLEX` for `cladiv`
`COMPLEX*16` for `zladiv`
The complex scalars x and y (for complex flavors only).

Output Parameters

p, q — `REAL` for `sladiv`
`DOUBLE PRECISION` for `dladiv`
The scalars p and q in the above expression (for real flavors only).

res — `COMPLEX` for `cladiv`
`DOUBLE COMPLEX` for `zladiv`
Contains the result of division x / y.

?lae2

Computes the eigenvalues of a 2-by-2 symmetric matrix.

Syntax

```
call slae2 ( a, b, c, rt1, rt2 )
call dlae2 ( a, b, c, rt1, rt2 )
```

Description

The routines `sla2/dlae2` compute the eigenvalues of a 2-by-2 symmetric matrix

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix}$$

On return, *rt1* is the eigenvalue of larger absolute value, and *rt1* is the eigenvalue of smaller absolute value.

Input Parameters

a, b, c — `REAL` for `slae2`
`DOUBLE PRECISION` for `dlae2`
The elements *a, b,* and *c* of the 2-by-2 matrix above.

Output Parameters

rt1, rt2 — `REAL` for `slae2`
`DOUBLE PRECISION` for `dlae2`
The computed eigenvalues of larger and smaller absolute value, respectively.

Application Notes

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`.

?laebz

Computes the number of eigenvalues of a real symmetric tridiagonal matrix which are less than or equal to a given value, and performs other tasks required by the routine `?stebz`.

Syntax

```
call slaebz( ijob, nitmax, n, mmax, minp, nbmin, abstol,
    reltol, pivmin, d, e, e2, nval, ab, c, mout, nab,
    work, iwork, info )
call dlaebz( ijob, nitmax, n, mmax, minp, nbmin, abstol,
    reltol, pivmin, d, e, e2, nval, ab, c, mout, nab,
    work, iwork, info )
```

Description

The routine `?laebz` contains the iteration loops which compute and use the function *N*(*w*), which is the count of eigenvalues of a symmetric tridiagonal matrix *T* less than or equal to its argument *w*. It performs a choice of two types of loops:

ijob =1, followed by

ijob =2: It takes as input a list of intervals and returns a list of sufficiently small intervals whose union contains the same eigenvalues as the union of the original intervals. The input intervals are **(***ab*(j,1),*ab*(j,2)**]**, j=1,...,*minp*. The output interval **(***ab*(j,1),*ab*(j,2)**]** will contain eigenvalues *nab*(j,1)+1,...,*nab*(j,2), where 1 ≤ j ≤ *mout*.

ijob =3: It performs a binary search in each input interval **(***ab*(j,1),*ab*(j,2)**]** for a point *w*(j) such that *N*(*w*(j))=*nval*(j), and uses *c*(j) as the starting point of the search. If such a *w*(j) is found, then on output *ab*(j,1)=*ab*(j,2)=*w*. If no such *w*(j) is found, then on output **(***ab*(j,1),*ab*(j,2)**]** will be a small interval containing the point where *N*(*w*) jumps through *nval*(j), unless that point lies outside the initial interval.

Note that the intervals are in all cases half-open intervals, that is, of the form **(***a*,*b***]** , which includes *b* but not *a* .

To avoid underflow, the matrix should be scaled so that its largest element is no greater than *overflow*`**`(1/2) * *underflow*`**`(1/4) in absolute value. To assure the most accurate computation of small eigenvalues, the matrix should be scaled to be not much smaller than that, either.

Note: the arguments are, in general, **not** checked for unreasonable values.

Input Parameters

ijob — INTEGER. Specifies what is to be done:
= 1: Compute *nab* for the initial intervals.
= 2: Perform bisection iteration to find eigenvalues of *T*.
= 3: Perform bisection iteration to invert *N*(*w*), i.e., to find a point which has a specified number of eigenvalues of *T* to its left.
Other values will cause ?laebz to return with *info*=-1.

nitmax — INTEGER.
The maximum number of "levels" of bisection to be performed, i.e., an interval of width *W* will not be made smaller than 2^(-*nitmax*) * *W*. If not all intervals have converged after *nitmax* iterations, then *info* is set to the number of non-converged intervals.

n — INTEGER.
The dimension *n* of the tridiagonal matrix *T*. It must be at least 1.

mmax — INTEGER.
The maximum number of intervals. If more than *mmax* intervals are generated, then ?laebz will quit with *info*=*mmax*+1.

minp — INTEGER.
The initial number of intervals. It may not be greater than *mmax*.

nbmin — INTEGER.
The smallest number of intervals that should be processed using a vector loop. If zero, then only the scalar loop will be used.

abstol — REAL for slaebz
DOUBLE PRECISION for dlaebz.
The minimum (absolute) width of an interval. When an interval is narrower than *abstol*, or than *reltol* times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, i.e., converged. This must be at least zero.

reltol — REAL for slaebz
DOUBLE PRECISION for dlaebz.
The minimum relative width of an interval. When an interval is narrower than *abstol*, or than *reltol* times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, i.e., converged. Note: this should always be at least *radix*machine epsilon*.

pivmin REAL for `slaebz`
DOUBLE PRECISION for `dlaebz`.
The minimum absolute value of a "pivot" in the Sturm sequence loop. This **must** be at least
max |e(j)**2| * *safe_min* and at least *safe_min*, where *safe_min* is at least the smallest number that can divide one without overflow.

d, e, e2 REAL for `slaebz`
DOUBLE PRECISION for `dlaebz`.
Arrays, dimension (*n*) each.
The array *d* contains the diagonal elements of the tridiagonal matrix *T*.

The array *e* contains the off-diagonal elements of the tridiagonal matrix *T* in positions 1 through *n*-1. *e*(*n*) is arbitrary.

The array *e2* contains the squares of the off-diagonal elements of the tridiagonal matrix *T*. *e2*(*n*) is ignored.

nval INTEGER.
Array, dimension (*minp*).
If *ijob*=1 or 2, not referenced.
If *ijob*=3, the desired values of *N*(*w*).

ab REAL for `slaebz`
DOUBLE PRECISION for `dlaebz`.
Array, dimension (*mmax*,2)
The endpoints of the intervals. *ab*(j,1) is *a*(j), the left endpoint of the j-th interval, and *ab*(j,2) is *b*(j), the right endpoint of the j-th interval.

c REAL for `slaebz`
DOUBLE PRECISION for `dlaebz`.
Array, dimension (*mmax*)
If *ijob*=1, ignored.
If *ijob*=2, workspace.
If *ijob*=3, then on input *c*(j) should be initialized to the first search point in the binary search.

nab INTEGER.
Array, dimension (*mmax*,2)
If *ijob*=2, then on input, *nab*(i,j) should be set. It must satisfy the condition:
N(*ab*(i,1)) ≤ *nab*(i,1) ≤ *nab*(i,2) ≤ *N*(*ab*(i,2)), which means that in interval i only eigenvalues
nab(i,1)+1,...,*nab*(i,2) will be considered. Usually, *nab*(i,j)=*N*(*ab*(i,j)), from a previous call to `?laebz` with *ijob*=1.

	If *ijob*=3, normally, *nab* should be set to some distinctive value(s) before `?laebz` is called.
work	`REAL` for `slaebz` `DOUBLE PRECISION` for `dlaebz`. Workspace array, dimension (*mmax*).
iwork	`INTEGER`. Workspace array, dimension (*mmax*).

Output Parameters

nval	The elements of *nval* will be reordered to correspond with the intervals in *ab*. Thus, *nval*(j) on output will not, in general be the same as *nval*(j) on input, but it will correspond with the interval **(***ab*(j,1),*ab*(j,2)**]** on output.
ab	The input intervals will, in general, be modified, split, and reordered by the calculation.
mout	`INTEGER`. If *ijob*=1, the number of eigenvalues in the intervals. If *ijob*=2 or 3, the number of intervals output. If *ijob*=3, *mout* will equal *minp*.
nab	If *ijob*=1, then on output *nab*(i,j) will be set to *N*(*ab*(i,j)). If *ijob*=2, then on output, *nab*(i,j) will contain max(*na*(*k*),min(*nb*(*k*),*N*(*ab*(i,j)))), where *k* is the index of the input interval that the output interval **(***ab*(j,1),*ab*(j,2)**]** came from, and *na*(*k*) and *nb*(*k*) are the the input values of *nab*(*k*,1) and *nab*(*k*,2). If *ijob*=3, then on output, *nab*(i,j) contains *N*(*ab*(i,j)), unless *N*(*w*) > *nval*(i) for all search points w , in which case *nab*(i,1) will not be modified, i.e., the output value will be the same as the input value (modulo reorderings, see *nval* and *ab*), or unless *N*(*w*) < *nval*(i) for all search points *w* , in which case *nab*(i,2) will not be modified.
info	`INTEGER`. 0: All intervals converged. 1--*mmax*: The last *info* intervals did not converge. *mmax*+1: More than *mmax* intervals were generated.

Application Notes

This routine is intended to be called only by other LAPACK routines, thus the interface is less user-friendly. It is intended for two purposes:

(a) finding eigenvalues. In this case, `?laebz` should have one or more initial intervals set up in *ab*, and `?laebz` should be called with *ijob*=1. This sets up *nab*, and also counts the eigenvalues. Intervals with no eigenvalues would usually be thrown out at this point. Also, if not all the eigenvalues in an interval i are desired, *nab*(i,1) can be increased or *nab*(i,2) decreased. For example, set *nab*(i,1)=*nab*(i,2)-1 to get the largest eigenvalue. `?laebz` is then called with *ijob*=2 and *mmax* no smaller than the value of *mout* returned by the call with *ijob*=1. After this (*ijob*=2) call, eigenvalues *nab*(i,1)+1 through *nab*(i,2) are approximately *ab*(i,1) (or *ab*(i,2)) to the tolerance specified by *abstol* and *reltol*.

(b) finding an interval **(***a*',*b*'**]** containing eigenvalues *w*(f),...,*w*(l). In this case, start with a Gershgorin interval **(***a*,*b***)**. Set up *ab* to contain 2 search intervals, both initially **(***a*,*b***)**. One *nval* element should contain f-1 and the other should contain l, while *c* should contain *a* and *b*, respectively. *nab*(i,1) should be -1 and *nab*(i,2) should be *n*+1, to flag an error if the desired interval does not lie in **(***a*,*b***)**. `?laebz` is then called with *ijob*=3. On exit, if *w*(f-1) < *w*(f), then one of the intervals -- j -- will have *ab*(j,1)=*ab*(j,2) and *nab*(j,1)=*nab*(j,2)=f-1, while if, to the specified tolerance, *w*(f-k)=...=*w*(f+r), k > 0 and r ≥ 0, then the interval will have *N*(*ab*(j,1))=*nab*(j,1)=f-k and *N*(*ab*(j,2))=*nab*(j,2)=f+r. The cases *w*(l) < *w*(l+1) and *w*(l-r)=...=*w*(l+k) are handled similarly.

?laed0

Used by `?stedc`*. Computes all eigenvalues and corresponding eigenvectors of an unreduced symmetric tridiagonal matrix using the divide and conquer method.*

Syntax

```
call slaed0(icompq, qsiz, n, d, e, q, ldq, qstore, ldqs,
       work, iwork, info)
call dlaed0(icompq, qsiz, n, d, e, q, ldq, qstore, ldqs,
       work, iwork, info)
call claed0(qsiz, n, d, e, q, ldq, qstore, ldqs, rwork,
       iwork, info)
call zlaed0(qsiz, n, d, e, q, ldq, qstore, ldqs, rwork,
       iwork, info)
```

Description

Real flavors of this routine compute all eigenvalues and (optionally) corresponding eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.

Complex flavors `claed0/zlaed0` compute all eigenvalues of a symmetric tridiagonal matrix which is one diagonal block of those from reducing a dense or band Hermitian matrix and corresponding eigenvectors of the dense or band matrix.

Input Parameters

icompq — `INTEGER`. Used with real flavors only.
If *icompq* = 0, compute eigenvalues only.
If *icompq* = 1, compute eigenvectors of original dense symmetric matrix also. On entry, the array *q* must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.
If *icompq* = 2, compute eigenvalues and eigenvectors of the tridiagonal matrix.

qsiz — `INTEGER`.

The dimension of the orthogonal/unitary matrix used to reduce the full matrix to tridiagonal form; $qsiz \ge n$ (for real flavors, $qsiz \ge n$ if $icompq = 1$).

n — `INTEGER`. The dimension of the symmetric tridiagonal matrix ($n \ge 0$).

d, e, rwork — `REAL` for single-precision flavors
`DOUBLE PRECISION` for double-precision flavors.
Arrays:
d(*) contains the main diagonal of the tridiagonal matrix. The dimension of *d* must be at least max(1, *n*).

e(*) contains the off-diagonal elements of the tridiagonal matrix. The dimension of *e* must be at least max(1, *n*-1).

rwork(*) is a workspace array used in complex flavors only. The dimension of *rwork* must be at least
$(1 + 3n + 2n\lg(n) + 3n^2)$, where $\lg(n)$ = smallest integer k such that $2^k \ge n$.

q, qstore — `REAL` for `slaed0`
`DOUBLE PRECISION` for `dlaed0`
`COMPLEX` for `claed0`
`COMPLEX*16` for `zlaed0`.
Arrays: *q*(*ldq*, *), *qstore*(*ldqs*, *). The second dimension of these arrays must be at least max(1, *n*).
For real flavors:

If *icompq* = 0, array *q* is not referenced.
If *icompq* = 1, on entry, *q* is a subset of the columns of the orthogonal matrix used to reduce the full matrix to tridiagonal form corresponding to the subset of the full matrix which is being decomposed at this time.
If *icompq* = 2, on entry, *q* will be the identity matrix.
The array *qstore* is a workspace array referenced only when *icompq* = 1. Used to store parts of the eigenvector matrix when the updating matrix multiplies take place.

For complex flavors:
On entry, *q* must contain an *qsiz*-by-*n* matrix whose columns are unitarily orthonormal. It is a part of the unitary matrix that reduces the full dense Hermitian matrix to a (reducible) symmetric tridiagonal matrix.
The array *qstore* is a workspace array used to store parts of the eigenvector matrix when the updating matrix multiplies take place.

ldq — `INTEGER`. The first dimension of the array *q*;
ldq ≥ max(1, *n*).

ldqs — `INTEGER`. The first dimension of the array *qstore*; *ldqs* ≥ max(1, *n*).

work — `REAL` for `slaed0`
`DOUBLE PRECISION` for `dlaed0`.
Workspace array, used in real flavors only.
If *icompq* = 0 or 1, the dimension of *work* must be at least $(1+3n+2n\lg(n)+2n^2)$, where $\lg(n)$ = smallest integer k such that $2^k \ge n$.
If *icompq* = 2, the dimension of *work* must be at least $(4n+n^2)$.

iwork — `INTEGER`.
Workspace array.
For real flavors, if *icompq* = 0 or 1, and for complex flavors, the dimension of *iwork* must be at least
$(6 +6n+5n\lg(n))$,
For real flavors, If *icompq* = 2, the dimension of *iwork* must be at least $(3+5n)$.

Output Parameters

d — On exit, contains eigenvalues in ascending order.

e — On exit, the array has been destroyed.

q — If *icompq* = 2, on exit, *q* contains the eigenvectors of the tridiagonal matrix.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i* > 0, the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *i*/(*n*+1) through mod(*i*, *n*+1).

?laed1

Used by sstedc/dstedc. *Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is tridiagonal.*

Syntax

```
call slaed1( n, d, q, ldq, indxq, rho, cutpnt, work,
        iwork, info)
call dlaed1( n, d, q, ldq, indxq, rho, cutpnt, work,
        iwork, info)
```

Description

The routine ?laed1 computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. This routine is used only for the eigenproblem which requires all eigenvalues and eigenvectors of a tridiagonal matrix. ?laed7 handles the case in which eigenvalues only or eigenvalues and eigenvectors of a full symmetric matrix (which was reduced to tridiagonal form) are desired.

$T = Q(\text{in}) (D(\text{in}) + rho * z*z') Q'(\text{in}) = Q(\text{out}) * D(\text{out}) * Q'(\text{out})$

where $z = Q'u$, u is a vector of length *n* with ones in the *cutpnt* and (*cutpnt* + 1) -th elements and zeros elsewhere. The eigenvectors of the original matrix are stored in *Q*, and the eigenvalues are in *D*. The algorithm consists of three stages:

The first stage consists of deflating the size of the problem when there are multiple eigenvalues or if there is a zero in the z vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine ?laed2.

The second stage consists of calculating the updated eigenvalues. This is done by finding the roots of the secular equation via the routine `?laed4` (as called by `?laed3`). This routine also calculates the eigenvectors of the current problem.

The final stage consists of computing the updated eigenvectors directly using the updated eigenvalues. The eigenvectors for the current problem are multiplied with the eigenvectors from the overall problem.

Input Parameters

n	`INTEGER`. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).
d, q, work	`REAL` for `slaed1` `DOUBLE PRECISION` for `dlaed1`. Arrays: *d*(*) contains the eigenvalues of the rank-1-perturbed matrix. The dimension of *d* must be at least max(1, *n*). *q*(*ldq*, *) contains the eigenvectors of the rank-1-perturbed matrix. The second dimension of *q* must be at least max(1, *n*). *work*(*) is a workspace array, dimension at least ($4n+n^2$).
ldq	`INTEGER`. The first dimension of the array *q*; $ldq \geq \max(1, n)$.
indxq	`INTEGER`. Array, dimension (*n*). On entry, the permutation which separately sorts the two subproblems in *d* into ascending order.
rho	`REAL` for `slaed1` `DOUBLE PRECISION` for `dlaed1`. The subdiagonal entry used to create the rank-1 modification.
cutpnt	`INTEGER`. The location of the last eigenvalue in the leading sub-matrix. $\min(1,n) \leq cutpnt \leq n/2$.
iwork	`INTEGER`. Workspace array, dimension (4*n*).

Output Parameters

d	On exit, contains the eigenvalues of the repaired matrix.
q	On exit, *q* contains the eigenvectors of the repaired tridiagonal matrix.

indxq On exit, contains the permutation which will reintegrate the subproblems back into sorted order, that is,
d(*indxq*(i = 1, *n*)) will be in ascending order.

info `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = 1, an eigenvalue did not converge.

?laed2

Used by `sstedc/dstedc`. *Merges eigenvalues and deflates secular equation. Used when the original matrix is tridiagonal.*

Syntax

```
call slaed2( k, n, n1, d, q, ldq, indxq, rho, z, dlamda,
        w, q2, indx, indxc, indxp, coltyp, info)
call dlaed2( k, n, n1, d, q, ldq, indxq, rho, z, dlamda,
        w, q2, indx, indxc, indxp, coltyp, info)
```

Description

The routine `?laed2` merges the two sets of eigenvalues together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more eigenvalues are close together or if there is a tiny entry in the *z* vector. For each such occurrence the order of the related secular equation problem is reduced by one.

Input Parameters

k `INTEGER`. The number of non-deflated eigenvalues, and the order of the related secular equation $(0 \le k \le n)$.

n `INTEGER`. The dimension of the symmetric tridiagonal matrix $(n \ge 0)$.

n1 `INTEGER`. The location of the last eigenvalue in the leading sub-matrix; $\min(1,n) \le n1 \le n/2$.

d, q, z	`REAL` for `slaed2` `DOUBLE PRECISION` for `dlaed2`. Arrays: *d*(*) contains the eigenvalues of the two submatrices to be combined. The dimension of *d* must be at least max(1, *n*). *q*(*ldq*, *) contains the eigenvectors of the two submatrices in the two square blocks with corners at (1,1), (*n1*,*n1*) and (*n1*+1,*n1*+1), (*n*,*n*). The second dimension of *q* must be at least max(1, *n*). *z*(*) contains the updating vector (the last row of the first sub-eigenvector matrix and the first row of the second sub-eigenvector matrix).
ldq	`INTEGER`. The first dimension of the array *q*; $ldq \geq \max(1, n)$.
indxq	`INTEGER`. Array, dimension (*n*). On entry, the permutation which separately sorts the two subproblems in *d* into ascending order. Note that elements in the second half of this permutation must first have *n1* added to their values.
rho	`REAL` for `slaed2` `DOUBLE PRECISION` for `dlaed2`. On entry, the off-diagonal element associated with the rank-1 cut which originally split the two submatrices which are now being recombined.
indx, indxp	`INTEGER`. Workspace arrays, dimension (*n*) each. Array *indx* contains the permutation used to sort the contents of *dlamda* into ascending order. Array *indxp* contains the permutation used to place deflated values of *d* at the end of the array. *indxp*(1:*k*) points to the nondeflated *d*-values and *indxp*(*k*+1:*n*) points to the deflated eigenvalues.
coltyp	`INTEGER`. Workspace array, dimension (*n*). During execution, a label which will indicate which of the following types a column in the *q2* matrix is: 1 : non-zero in the upper half only; 2 : dense; 3 : non-zero in the lower half only; 4 : deflated.

Output Parameters

d On exit, *d* contains the trailing (*n*-*k*) updated eigenvalues (those which were deflated) sorted into increasing order.

q On exit, *q* contains the trailing (*n*-*k*) updated eigenvectors (those which were deflated) in its last *n*-*k* columns.

indxq Destroyed on exit.

rho On exit, *rho* has been modified to the value required by `?laed3`.

dlamda, w, q2 `REAL` for `slaed2`
`DOUBLE PRECISION` for `dlaed2`.
Arrays: *dlamda*(*n*), *w*(*n*), *q2*($n1^2$+$(n-n1)^2$).

The array *dlamda* contains a copy of the first *k* eigenvalues which will be used by `?laed3` to form the secular equation.

The array *w* contains the first *k* values of the final deflation-altered *z*-vector which will be passed to `?laed3`.

The array *q2* contains a copy of the first *k* eigenvectors which will be used by `?laed3` in a matrix multiply (`sgemm/dgemm`) to solve for the new eigenvectors.

indxc `INTEGER`. Array, dimension (*n*).
The permutation used to arrange the columns of the deflated *q* matrix into three groups: the first group contains non-zero elements only at and above *n1*, the second contains non-zero elements only below *n1*, and the third is dense.

coltyp On exit, *coltyp*(i) is the number of columns of type i, for i=1 to 4 only (see the definition of types in the description of *coltyp* in *Input Parameters*).

info `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

?laed3

Used by `sstedc/dstedc`. *Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is tridiagonal.*

Syntax

```
call slaed3( k, n, n1, d, q, ldq, rho, dlamda, q2, indx,
       ctot, w, s, info)
call dlaed3( k, n, n1, d, q, ldq, rho, dlamda, q2, indx,
       ctot, w, s, info)
```

Description

The routine `?laed3` finds the roots of the secular equation, as defined by the values in *d*, *w*, and *rho*, between 1 and *k*. It makes the appropriate calls to `?laed4` and then updates the eigenvectors by multiplying the matrix of eigenvectors of the pair of eigensystems being combined by the matrix of eigenvectors of the *k*-by-*k* system which is solved here.

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but none are known.

Input Parameters

k — `INTEGER`. The number of terms in the rational function to be solved by `?laed4` ($k \geq 0$).

n — `INTEGER`. The number of rows and columns in the *q* matrix. $n \geq k$ (deflation may result in $n > k$).

n1 — `INTEGER`. The location of the last eigenvalue in the leading sub-matrix; $\min(1,n) \leq n1 \leq n/2$.

q — `REAL` for `slaed3`
`DOUBLE PRECISION` for `dlaed3`.
Array *q*(*ldq*, *). The second dimension of *q* must be at least max(1, *n*). Initially, the first *k* columns of this array are used as workspace.

ldq — `INTEGER`. The first dimension of the array *q*; $ldq \geq \max(1, n)$.

rho	`REAL` for `slaed3` `DOUBLE PRECISION` for `dlaed3`. The value of the parameter in the rank one update equation. *rho* ≥ 0 required.
dlamda, q2, w	`REAL` for `slaed3` `DOUBLE PRECISION` for `dlaed3`. Arrays: *dlamda*(*k*), *q2*(*ldq2*, *), *w*(*k*). The first *k* elements of the array *dlamda* contain the old roots of the deflated updating problem. These are the poles of the secular equation. The first *k* columns of the array *q2* contain the non-deflated eigenvectors for the split problem. The second dimension of *q2* must be at least max(1, *n*). The first *k* elements of the array *w* contain the components of the deflation-adjusted updating vector.
indx	`INTEGER`. Array, dimension (*n*). The permutation used to arrange the columns of the deflated *q* matrix into three groups (see `?laed2`). The rows of the eigenvectors found by `?laed4` must be likewise permuted before the matrix multiply can take place.
ctot	`INTEGER`. Array, dimension (4). A count of the total number of the various types of columns in *q* , as described in *indx*. The fourth column type is any column which has been deflated.
s	`REAL` for `slaed3` `DOUBLE PRECISION` for `dlaed3`. Workspace array, dimension (*n1*+1)**k* . Will contain the eigenvectors of the repaired matrix which will be multiplied by the previously accumulated eigenvectors to update the system.

Output Parameters

d	`REAL` for `slaed3` `DOUBLE PRECISION` for `dlaed3`. Array, dimension at least max(1, *n*). *d*(i) contains the updated eigenvalues for 1 ≤ i ≤ *k*.
q	On exit, the columns 1 to *k* of *q* contain the updated eigenvectors.
dlamda	May be changed on output by having lowest order bit set to zero on Cray X-MP, Cray Y-MP, Cray-2, or Cray C-90, as described above.
w	Destroyed on exit.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = 1, an eigenvalue did not converge.

?laed4

Used by sstedc/dstedc. *Finds a single root of the secular equation.*

Syntax

```
call slaed4 ( n, i, d, z, delta, rho, dlam, info )
call dlaed4 ( n, i, d, z, delta, rho, dlam, info )
```

Description

This subroutine computes the *i*-th updated eigenvalue of a symmetric rank-one modification to a diagonal matrix whose elements are given in the array *d*, and that

$D(i) < D(j)$ for $i < j$

and that *rho* > 0. This is arranged by the calling routine, and is no loss in generality. The rank-one modified system is thus

diag(*D*) + *rho* * *Z* * transpose(*Z*) .

where we assume the Euclidean norm of *Z* is 1.

The method consists of approximating the rational functions in the secular equation by simpler interpolating rational functions.

Input Parameters

n INTEGER. The length of all arrays.

i INTEGER. The index of the eigenvalue to be computed; $1 \le i \le n$.

d, z — REAL for `slaed4`
DOUBLE PRECISION for `dlaed4`
Arrays, dimension (*n*) each.
The array *d* contains the original eigenvalues. It is assumed that they are in order, *d*(i) < *d*(j) for i < j.

The array *z* contains the components of the updating vector *Z*.

rho — REAL for `slaed4`
DOUBLE PRECISION for `dlaed4`
The scalar in the symmetric updating formula.

Output Parameters

delta — REAL for `slaed4`
DOUBLE PRECISION for `dlaed4`
Array, dimension (*n*).
If $n \neq 1$, *delta* contains (*d*(j) - *lambda_i*) in its j-th component. If *n* = 1, then *delta*(1) = 1. The vector *delta* contains the information necessary to construct the eigenvectors.

dlam — REAL for `slaed4`
DOUBLE PRECISION for `dlaed4`
The computed *lambda_i*, the *i*-th updated eigenvalue.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = 1, the updating process failed.

?laed5

Used by `sstedc/dstedc`.
Solves the 2-by-2 secular equation.

Syntax

```
call slaed5 ( i, d, z, delta, rho, dlam )
call dlaed5 ( i, d, z, delta, rho, dlam )
```

Description

This subroutine computes the `i`-th eigenvalue of a symmetric rank-one modification of a 2-by-2 diagonal matrix

$$\mathrm{diag}(D) + rho * Z * \mathrm{transpose}(Z) .$$

The diagonal elements in the array *D* are assumed to satisfy

$$D(\mathrm{i}) < D(\mathrm{j}) \ \text{ for } \ \mathrm{i} < \mathrm{j} .$$

We also assume `rho` > 0 and that the Euclidean norm of the vector *Z* is one.

Input Parameters

i	`INTEGER`. The index of the eigenvalue to be computed; $1 \le \mathrm{i} \le 2$.
d, z	`REAL` for `slaed5` `DOUBLE PRECISION` for `dlaed5` Arrays, dimension (2) each. The array *d* contains the original eigenvalues. It is assumed that *d*(1) < *d*(2). The array *z* contains the components of the updating vector.
rho	`REAL` for `slaed5` `DOUBLE PRECISION` for `dlaed5` The scalar in the symmetric updating formula.

Output Parameters

delta	`REAL` for `slaed5` `DOUBLE PRECISION` for `dlaed5` Array, dimension (2). The vector *delta* contains the information necessary to construct the eigenvectors.
dlam	`REAL` for `slaed5` `DOUBLE PRECISION` for `dlaed5` The computed *lambda_i*, the `i`-th updated eigenvalue.

?laed6

Used by `sstedc/dstedc`.
Computes one Newton step in solution of the secular equation.

Syntax

```
call slaed6(kniter, orgati, rho, d, z, finit, tau, info)
call dlaed6(kniter, orgati, rho, d, z, finit, tau, info)
```

Description

This routine computes the positive or negative root (closest to the origin) of

$$f(x) = rho + \frac{z(1)}{d(1) - x} + \frac{z(2)}{d(2) - x} + \frac{z(3)}{d(3) - x}$$

It is assumed that if *orgati* = `.TRUE.` the root is between $d(2)$ and $d(3)$; otherwise it is between $d(1)$ and $d(2)$
This routine will be called by `?laed4` when necessary. In most cases, the root sought is the smallest in magnitude, though it might not be in some extremely rare situations.

Input Parameters

kniter — `INTEGER`.
Refer to `?laed4` for its significance.

orgati — `LOGICAL`.
If *orgati* = `.TRUE.`, the needed root is between $d(2)$ and $d(3)$; otherwise it is between $d(1)$ and $d(2)$. See `?laed4` for further details.

rho — `REAL` for `slaed6`
`DOUBLE PRECISION` for `dlaed6`
Refer to the equation for *f(x)* above.

d, z — `REAL` for `slaed6`
`DOUBLE PRECISION` for `dlaed6`
Arrays, dimension (3) each.

The array *d* satisfies $d(1) < d(2) < d(3)$.

Each of the elements in the array *z* must be positive.

finit — REAL for slaed6
DOUBLE PRECISION for dlaed6
The value of *f(x)* at 0. It is more accurate than the one evaluated inside this routine (if someone wants to do so).

Output Parameters

tau — REAL for slaed6
DOUBLE PRECISION for dlaed6
The root of the equation for *f(x)*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = 1, failure to converge.

?laed7

Used by ?stedc. *Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is dense.*

Syntax

```
call slaed7( icompq, n, qsiz, tlvls, curlvl, curpbm, d, q, ldq,
      indxq, rho, cutpnt, qstore, qptr, prmptr, perm, givptr, givcol,
      givnum, work, iwork, info )
call dlaed7( icompq, n, qsiz, tlvls, curlvl, curpbm, d, q, ldq,
      indxq, rho, cutpnt, qstore, qptr, prmptr, perm, givptr, givcol,
      givnum, work, iwork, info )
call claed7( n, cutpnt, qsiz, tlvls, curlvl, curpbm, d, q, ldq, rho,
     indxq, qstore, qptr, prmptr, perm, givptr, givcol, givnum,
      work, rwork, iwork, info )
call zlaed7( n, cutpnt, qsiz, tlvls, curlvl, curpbm, d, q, ldq, rho,
     indxq, qstore, qptr, prmptr, perm, givptr, givcol, givnum,
      work, rwork, iwork, info )
```

Description

The routine `?laed7` computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. This routine is used only for the eigenproblem which requires all eigenvalues and optionally eigenvectors of a dense symmetric/Hermitian matrix that has been reduced to tridiagonal form. For real flavors, `slaed1/dlaed1` handles the case in which all eigenvalues and eigenvectors of a symmetric tridiagonal matrix are desired.

T = *Q*(in) (*D*(in) + *rho* * z*z') *Q*'(in) = *Q*(out) * *D*(out) * *Q*'(out)

where z = *Q*'*u*, *u* is a vector of length *n* with ones in the *cutpnt* and (*cutpnt* + 1) -th elements and zeros elsewhere. The eigenvectors of the original matrix are stored in *Q*, and the eigenvalues are in *D*. The algorithm consists of three stages:

The first stage consists of deflating the size of the problem when there are multiple eigenvalues or if there is a zero in the z vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine `slaed8/dlaed8` (for real flavors) or by the routine `slaed2/dlaed2` (for complex flavors) .

The second stage consists of calculating the updated eigenvalues. This is done by finding the roots of the secular equation via the routine `?laed4` (as called by `?laed9` or `?laed3`). This routine also calculates the eigenvectors of the current problem.

The final stage consists of computing the updated eigenvectors directly using the updated eigenvalues. The eigenvectors for the current problem are multiplied with the eigenvectors from the overall problem.

Input Parameters

icompq — `INTEGER`. Used with real flavors only.
If *icompq* = 0, compute eigenvalues only.
If *icompq* = 1, compute eigenvectors of original dense symmetric matrix also. On entry, the array *q* must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.

n — `INTEGER`. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).

cutpnt — `INTEGER`. The location of the last eigenvalue in the leading sub-matrix. $\min(1,n) \leq cutpnt \leq n$.

qsiz — `INTEGER`. The dimension of the orthogonal/unitary matrix used to reduce the full matrix to tridiagonal form; $qsiz \geq n$ (for real flavors, $qsiz \geq n$ if *icompq* = 1).

tlvls	`INTEGER`. The total number of merging levels in the overall divide and conquer tree.
curlvl	`INTEGER`. The current level in the overall merge routine, $0 \le curlvl \le tlvls$.
curpbm	`INTEGER`. The current problem in the current level in the overall merge routine (counting from upper left to lower right).
d	`REAL` for `slaed7/claed7` `DOUBLE PRECISION` for `dlaed7/zlaed7`. Array, dimension at least max(1, *n*). Array *d*(*) contains the eigenvalues of the rank-1-perturbed matrix.
q, work	`REAL` for `slaed7` `DOUBLE PRECISION` for `dlaed7` `COMPLEX` for `claed7` `COMPLEX*16` for `zlaed7`. Arrays: *q*(*ldq*, *) contains the the eigenvectors of the rank-1-perturbed matrix. The second dimension of *q* must be at least max(1, *n*). *work*(*) is a workspace array, dimension at least (3*n*+*qsiz***n*) for real flavors and at least (*qsiz***n*) for complex flavors.
ldq	`INTEGER`. The first dimension of the array *q*; $ldq \ge \max(1, n)$.
rho	`REAL` for `slaed7/claed7` `DOUBLE PRECISION` for `dlaed7/zlaed7`. The subdiagonal element used to create the rank-1 modification.
qstore	`REAL` for `slaed7/claed7` `DOUBLE PRECISION` for `dlaed7/zlaed7`. Array, dimension (n^2+1). Serves also as output parameter. Stores eigenvectors of submatrices encountered during divide and conquer, packed together. *qptr* points to beginning of the submatrices.
qptr	`INTEGER`. Array, dimension (*n*+2). Serves also as output parameter. List of indices pointing to beginning of submatrices stored in *qstore*. The submatrices are numbered starting at the bottom left of the divide and conquer tree, from left to right and bottom to top.
prmptr, perm, givptr	`INTEGER`. Arrays, dimension ($n \lg n$) each.

The array *prmptr*(*) contains a list of pointers which indicate where in *perm* a level's permutation is stored. *prmptr*(i+1) - *prmptr*(i) indicates the size of the permutation and also the size of the full, non-deflated problem.

The array *perm*(*) contains the permutations (from deflation and sorting) to be applied to each eigenblock.

The array *givptr*(*) contains a list of pointers which indicate where in *givcol* a level's Givens rotations are stored. *givptr*(i+1) - *givptr*(i) indicates the number of Givens rotations.

givcol INTEGER. Array, dimension (2, *n* lg*n*).
Each pair of numbers indicates a pair of columns to take place in a Givens rotation.

givnum REAL for slaed7/claed7
DOUBLE PRECISION for dlaed7/zlaed7.
Array, dimension (2, *n* lg*n*).
Each number indicates the *S* value to be used in the corresponding Givens rotation.

iwork INTEGER. Workspace array, dimension (4*n*).

rwork REAL for claed7
DOUBLE PRECISION for zlaed7.
Workspace array, dimension (3*n*+2*qsiz**n*). Used in complex flavors only.

Output Parameters

d On exit, contains the eigenvalues of the repaired matrix.

q On exit, *q* contains the eigenvectors of the repaired tridiagonal matrix.

indxq INTEGER. Array, dimension (*n*).
Contains the permutation which will reintegrate the subproblems back into sorted order, that is,
d(*indxq*(i = 1, *n*)) will be in ascending order.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = 1, an eigenvalue did not converge.

?laed8

Used by `?stedc`*. Merges eigenvalues and deflates secular equation. Used when the original matrix is dense.*

Syntax

```
call slaed8( icompq, k, n, qsiz, d, q, ldq, indxq, rho, cutpnt, z,
      dlamda, q2, ldq2, w, perm, givptr, givcol, givnum, indxp, indx,
      info )
call dlaed8( icompq, k, n, qsiz, d, q, ldq, indxq, rho, cutpnt, z,
      dlamda, q2, ldq2, w, perm, givptr, givcol, givnum, indxp, indx,
      info )
call claed8( k, n, qsiz, q, ldq, d, rho, cutpnt, z, dlamda, q2,
      ldq2, w, indxp, indx, indxq, perm, givptr, givcol, givnum,
      info )
call zlaed8( k, n, qsiz, q, ldq, d, rho, cutpnt, z, dlamda, q2,
      ldq2, w, indxp, indx, indxq, perm, givptr, givcol, givnum,
      info )
```

Description

This routine merges the two sets of eigenvalues together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more eigenvalues are close together or if there is a tiny element in the `z` vector. For each such occurrence the order of the related secular equation problem is reduced by one.

Input Parameters

icompq — `INTEGER`. Used with real flavors only.
If *icompq* = 0, compute eigenvalues only.
If *icompq* = 1, compute eigenvectors of original dense symmetric matrix also. On entry, the array *q* must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.

n — `INTEGER`. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).

cutpnt — `INTEGER`. The location of the last eigenvalue in the leading sub-matrix. $\min(1,n) \leq cutpnt \leq n$.

qsiz — INTEGER. The dimension of the orthogonal/unitary matrix used to reduce the full matrix to tridiagonal form; $qsiz \geq n$ (for real flavors, $qsiz \geq n$ if $icompq = 1$).

d, z — REAL for slaed8/claed8
DOUBLE PRECISION for dlaed8/zlaed8.
Arrays, dimension at least max(1, *n*) each.
The array *d*(*) contains the eigenvalues of the two submatrices to be combined.
On entry, *z*(*) contains the updating vector (the last row of the first sub-eigenvector matrix and the first row of the second sub-eigenvector matrix).
The contents of *z* are destroyed by the updating process.

q — REAL for slaed8
DOUBLE PRECISION for dlaed8
COMPLEX for claed8
COMPLEX*16 for zlaed8.
Array *q*(*ldq*, *). The second dimension of *q* must be at least max(1, *n*). On entry, *q* contains the eigenvectors of the partially solved system which has been previously updated in matrix multiplies with other partially solved eigensystems.
For real flavors, if $icompq = 0$, *q* is not referenced.

ldq — INTEGER. The first dimension of the array *q*;
$ldq \geq \max(1, n)$.

ldq2 — INTEGER. The first dimension of the output array *q2*;
$ldq2 \geq \max(1, n)$.

indxq — INTEGER. Array, dimension (*n*).
The permutation which separately sorts the two sub-problems in *d* into ascending order. Note that elements in the second half of this permutation must first have *cutpnt* added to their values in order to be accurate.

rho — REAL for slaed8/claed8
DOUBLE PRECISION for dlaed8/zlaed8.
On entry, the off-diagonal element associated with the rank-1 cut which originally split the two submatrices which are now being recombined.

Output Parameters

k — INTEGER. The number of non-deflated eigenvalues, and the order of the related secular equation.

d — On exit, contains the trailing (*n*-*k*) updated eigenvalues (those which were deflated) sorted into increasing order.

q — On exit, *q* contains the trailing (*n*-*k*) updated eigenvectors (those which were deflated) in its last (*n*-*k*) columns.

rho — On exit, *rho* has been modified to the value required by `?laed3`.

dlamda, w — `REAL` for `slaed8/claed8`
`DOUBLE PRECISION` for `dlaed8/zlaed8`.
Arrays, dimension (*n*) each.
The array *dlamda*(*) contains a copy of the first *k* eigenvalues which will be used by `?laed3` to form the secular equation.

The array *w*(*) will hold the first *k* values of the final deflation-altered z-vector and will be passed to `?laed3`.

q2 — `REAL` for `slaed8`
`DOUBLE PRECISION` for `dlaed8`
`COMPLEX` for `claed8`
`COMPLEX*16` for `zlaed8`.
Array *q2*(*ldq2*, *). The second dimension of *q2* must be at least max(1, *n*).
Contains a copy of the first *k* eigenvectors which will be used by `slaed7/dlaed7` in a matrix multiply (`sgemm/dgemm`) to update the new eigenvectors.
For real flavors, if *icompq* = 0, *q2* is not referenced.

indxp, indx — `INTEGER`. Workspace arrays, dimension (*n*) each.

The array *indxp*(*) will contain the permutation used to place deflated values of *d* at the end of the array. On output, *indxp*(1:*k*) points to the nondeflated *d*-values and *indxp*(*k*+1:*n*) points to the deflated eigenvalues.

The array *indx*(*) will contain the permutation used to sort the contents of *d* into ascending order.

perm — `INTEGER`. Array, dimension (*n*).
Contains the permutations (from deflation and sorting) to be applied to each eigenblock.

givptr — `INTEGER`. Contains the number of Givens rotations which took place in this subproblem.

givcol — `INTEGER`. Array, dimension (2, *n*).
Each pair of numbers indicates a pair of columns to take place in a Givens rotation.

givnum — REAL for slaed8/claed8
DOUBLE PRECISION for dlaed8/zlaed8.
Array, dimension (2, *n*).
Each number indicates the *S* value to be used in the corresponding Givens rotation.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

?laed9

Used by sstedc/dstedc.
Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is dense.

Syntax

```
call slaed9( k, kstart, kstop, n, d, q, ldq, rho,
      dlamda, w, s, lds, info )
call dlaed9( k, kstart, kstop, n, d, q, ldq, rho,
      dlamda, w, s, lds, info )
```

Description

This routine finds the roots of the secular equation, as defined by the values in *d*, z, and *rho*, between *kstart* and *kstop*. It makes the appropriate calls to slaed4/dlaed4 and then stores the new matrix of eigenvectors for use in calculating the next level of z vectors.

Input Parameters

k — INTEGER. The number of terms in the rational function to be solved by slaed4/dlaed4 ($k \geq 0$).

kstart, kstop — INTEGER. The updated eigenvalues *lambda*(i), $kstart \leq i \leq kstop$ are to be computed.
$1 \leq kstart \leq kstop \leq k$.

n — INTEGER. The number of rows and columns in the *Q* matrix. $n \geq k$ (deflation may result in $n > k$).

q — `REAL` for `slaed9`
`DOUBLE PRECISION` for `dlaed9`.
Workspace array, dimension (*ldq*, *). The second dimension of *q* must be at least max(1, *n*).

ldq — `INTEGER`. The first dimension of the array *q*;
ldq ≥ max(1, *n*).

rho — `REAL` for `slaed9`
`DOUBLE PRECISION` for `dlaed9`
The value of the parameter in the rank one update equation. *rho* ≥ 0 required.

dlamda, w — `REAL` for `slaed9`
`DOUBLE PRECISION` for `dlaed9`
Arrays, dimension (*k*) each.
The first *k* elements of the array *dlamda*(*) contain the old roots of the deflated updating problem. These are the poles of the secular equation.

The first *k* elements of the array *w*(*) contain the components of the deflation-adjusted updating vector.

lds — `INTEGER`. The first dimension of the output array *s*;
lds ≥ max(1, *k*).

Output Parameters

d — `REAL` for `slaed9`
`DOUBLE PRECISION` for `dlaed9`
Array, dimension (*n*). *d*(i) contains the updated eigenvalues for *kstart* ≤ i ≤ *kstop*.

s — `REAL` for `slaed9`
`DOUBLE PRECISION` for `dlaed9`.
Array, dimension (*lds*, *). The second dimension of *s* must be at least max(1, *k*).
Will contain the eigenvectors of the repaired matrix which will be stored for subsequent z vector calculation and multiplied by the previously accumulated eigenvectors to update the system.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.
If *info* = 1, the eigenvalue did not converge.

?laeda

Used by `?stedc`*. Computes the Z vector determining the rank-one modification of the diagonal matrix. Used when the original matrix is dense.*

Syntax

```
call slaeda( n, tlvls, curlvl, curpbm, prmptr, perm, givptr, givcol,
      givnum, q, qptr, z, ztemp, info )
call dlaeda( n, tlvls, curlvl, curpbm, prmptr, perm, givptr, givcol,
      givnum, q, qptr, z, ztemp, info )
```

Description

The routine `?laeda` computes the `z` vector corresponding to the merge step in the *curlvl*-th step of the merge process with *tlvls* steps for the *curpbm*-th problem.

Input Parameters

n — `INTEGER`. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).

tlvls — `INTEGER`. The total number of merging levels in the overall divide and conquer tree.

curlvl — `INTEGER`. The current level in the overall merge routine, $0 \leq curlvl \leq tlvls$.

curpbm — `INTEGER`. The current problem in the current level in the overall merge routine (counting from upper left to lower right).

prmptr, perm, givptr — `INTEGER`. Arrays, dimension (n lg$n$) each.

The array *prmptr*(*) contains a list of pointers which indicate where in *perm* a level's permutation is stored. *prmptr*(i+1) - *prmptr*(i) indicates the size of the permutation and also the size of the full, non-deflated problem.

The array *perm*(*) contains the permutations (from deflation and sorting) to be applied to each eigenblock.

The array *givptr*(*) contains a list of pointers which indicate where in *givcol* a level's Givens rotations are stored. *givptr*(i+1) - *givptr*(i) indicates the number of Givens rotations.

givcol — INTEGER. Array, dimension (2, *n* lg*n*).
Each pair of numbers indicates a pair of columns to take place in a Givens rotation.

givnum — REAL for slaeda
DOUBLE PRECISION for dlaeda.
Array, dimension (2, *n* lg*n*).
Each number indicates the *S* value to be used in the corresponding Givens rotation.

q — REAL for slaeda
DOUBLE PRECISION for dlaeda.
Array, dimension (n^2).
Contains the square eigenblocks from previous levels, the starting positions for blocks are given by *qptr*.

qptr — INTEGER. Array, dimension (*n*+2). Contains a list of pointers which indicate where in *q* an eigenblock is stored. sqrt(*qptr*(i+1) - *qptr*(i)) indicates the size of the block.

ztemp — REAL for slaeda
DOUBLE PRECISION for dlaeda.
Workspace array, dimension (*n*).

Output Parameters

z — REAL for slaeda
DOUBLE PRECISION for dlaeda.
Array, dimension (*n*). Contains the updating vector (the last row of the first sub-eigenvector matrix and the first row of the second sub-eigenvector matrix).

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*th parameter had an illegal value.

?laein

Computes a specified right or left eigenvector of an upper Hessenberg matrix by inverse iteration.

Syntax

```
call slaein( rightv, noinit, n, h, ldh, wr, wi, vr, vi, b, ldb,
        work, eps3, smlnum, bignum, info )
call dlaein( rightv, noinit, n, h, ldh, wr, wi, vr, vi, b, ldb,
        work, eps3, smlnum, bignum, info )
call claein( rightv, noinit, n, h, ldh, w, v, b, ldb,
        rwork, eps3, smlnum, info )
call zlaein( rightv, noinit, n, h, ldh, w, v, b, ldb,
        rwork, eps3, smlnum, info )
```

Description

The routine `?laein` uses inverse iteration to find a right or left eigenvector corresponding to the eigenvalue (*wr*,*wi*) of a real upper Hessenberg matrix *H* (for real flavors `slaein/dlaein`) or to the eigenvalue *w* of a complex upper Hessenberg matrix *H* (for complex flavors `claein/zlaein`).

Input Parameters

rightv — `LOGICAL`.
If *rightv* = `.TRUE.`, compute right eigenvector;
if *rightv* = `.FALSE.`, compute left eigenvector.

noinit — `LOGICAL`.
If *noinit* = `.TRUE.`, no initial vector is supplied in (*vr*,*vi*) or in *v* (for complex flavors);
if *noinit* = `.FALSE.`, initial vector is supplied in (*vr*,*vi*) or in *v* (for complex flavors).

n — `INTEGER`. The order of the matrix *H* ($n \ge 0$).

h — `REAL` for `slaein`
`DOUBLE PRECISION` for `dlaein`
`COMPLEX` for `claein`

	`COMPLEX*16` for `zlaein`. Array *h*(*ldh*, *). The second dimension of *h* must be at least max(1, *n*). Contains the upper Hessenberg matrix *H*.
ldh	`INTEGER`. The first dimension of the array *h*; *ldh* ≥ max(1, *n*).
wr, wi	`REAL` for `slaein` `DOUBLE PRECISION` for `dlaein`. The real and imaginary parts of the eigenvalue of *H* whose corresponding right or left eigenvector is to be computed (for real flavors of the routine).
w	`COMPLEX` for `claein` `COMPLEX*16` for `zlaein`. The eigenvalue of *H* whose corresponding right or left eigenvector is to be computed (for complex flavors of the routine).
vr, vi	`REAL` for `slaein` `DOUBLE PRECISION` for `dlaein`. Arrays, dimension (*n*) each. Used for real flavors only. On entry, if *noinit* = `.FALSE.` and *wi* = 0.0, *vr* must contain a real starting vector for inverse iteration using the real eigenvalue *wr*; if *noinit* = `.FALSE.` and *wi* ≠ 0.0, *vr* and *vi* must contain the real and imaginary parts of a complex starting vector for inverse iteration using the complex eigenvalue (*wr*,*wi*); otherwise *vr* and *vi* need not be set.
v	`COMPLEX` for `claein` `COMPLEX*16` for `zlaein`. Array, dimension (*n*). Used for complex flavors only. On entry, if *noinit* = `.FALSE.`, *v* must contain a starting vector for inverse iteration; otherwise *v* need not be set.
b	`REAL` for `slaein` `DOUBLE PRECISION` for `dlaein` `COMPLEX` for `claein` `COMPLEX*16` for `zlaein`. Workspace array *b*(*ldb*, *). The second dimension of *b* must be at least max(1, *n*).
ldb	`INTEGER`. The first dimension of the array *b*; *ldb* ≥ *n*+1 for real flavors; *ldb* ≥ max(1, *n*) for complex flavors.

work	`REAL` for `slaein` `DOUBLE PRECISION` for `dlaein`. Workspace array, dimension (*n*). Used for real flavors only.
rwork	`REAL` for `claein` `DOUBLE PRECISION` for `zlaein`. Workspace array, dimension (*n*). Used for complex flavors only.
eps3, smlnum	`REAL` for `slaein/claein` `DOUBLE PRECISION` for `dlaein/zlaein`. *eps3* is a small machine-dependent value which is used to perturb close eigenvalues, and to replace zero pivots. *smlnum* is a machine-dependent value close to underflow threshold.
bignum	`REAL` for `slaein` `DOUBLE PRECISION` for `dlaein`. *bignum* is a machine-dependent value close to overflow threshold. Used for real flavors only.

Output Parameters

vr, vi	On exit, if *wi* = 0.0 (real eigenvalue), *vr* contains the computed real eigenvector; if *wi* ≠ 0.0 (complex eigenvalue), *vr* and *vi* contain the real and imaginary parts of the computed complex eigenvector. The eigenvector is normalized so that the component of largest magnitude has magnitude 1; here the magnitude of a complex number (x,y) is taken to be $\|x\| + \|y\|$. *vi* is not referenced if *wi* = 0.0.
v	On exit, *v* contains the computed eigenvector, normalized so that the component of largest magnitude has magnitude 1; here the magnitude of a complex number (x,y) is taken to be $\|x\| + \|y\|$.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = 1, inverse iteration did not converge. For real flavors, *vr* is set to the last iterate, and so is *vi* if *wi* ≠ 0.0. For complex flavors, *v* is set to the last iterate.

?laev2

Computes the eigenvalues and eigenvectors of a 2-by-2 symmetric/Hermitian matrix.

Syntax

```
call slaev2 (a, b, c, rt1, rt2, cs1, sn1)
call dlaev2 (a, b, c, rt1, rt2, cs1, sn1)
call claev2 (a, b, c, rt1, rt2, cs1, sn1)
call zlaev2 (a, b, c, rt1, rt2, cs1, sn1)
```

Discussion

This routine performs the eigendecomposition of a 2-by-2 symmetric matrix

$\begin{bmatrix} a & b \\ b & c \end{bmatrix}$ (for `slaev2/dlaev2`) or Hermitian matrix $\begin{bmatrix} a & b \\ \mathrm{conjg}(b) & c \end{bmatrix}$

(for `claev2/zlaev2`).

On return, *rt1* is the eigenvalue of larger absolute value, *rt2* of smaller absolute value, and (*cs1*, *sn1*) is the unit right eigenvector for *rt1*, giving the decomposition

$$\begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} \cdot \begin{bmatrix} a & b \\ b & c \end{bmatrix} \cdot \begin{bmatrix} cs1 & -sn1 \\ sn1 & cs1 \end{bmatrix} = \begin{bmatrix} rt1 & 0 \\ 0 & rt2 \end{bmatrix}$$

(for `slaev2/dlaev2`),
or

$$\begin{bmatrix} cs1 & \mathrm{conjg}(sn1) \\ -sn1 & cs1 \end{bmatrix} \cdot \begin{bmatrix} a & b \\ \mathrm{conjg}(b) & c \end{bmatrix} \cdot \begin{bmatrix} cs1 & -\mathrm{conjg}(sn1) \\ sn1 & cs1 \end{bmatrix} = \begin{bmatrix} rt1 & 0 \\ 0 & rt2 \end{bmatrix}$$

(for `claev2/zlaev2`).

Input Parameters

a, b, c — REAL for slaev2
DOUBLE PRECISION for dlaev2
COMPLEX for claev2
COMPLEX*16 for zlaev2.
Elements of the input matrix.

Output Parameters

rt1, rt2 — REAL for slaev2/claev2
DOUBLE PRECISION for dlaev2/zlaev2.
Eigenvalues of larger and smaller absolute value, respectively.

cs1 — REAL for slaev2/claev2
DOUBLE PRECISION for dlaev2/zlaev2.

sn1 — REAL for slaev2
DOUBLE PRECISION for dlaev2
COMPLEX for claev2
COMPLEX*16 for zlaev2.
The vector (*cs1*, *sn1*) is the unit right eigenvector for *rt1*.

Application Notes

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.

?laexc

Swaps adjacent diagonal blocks of a real upper quasi-triangular matrix in Schur canonical form, by an orthogonal similarity transformation.

Syntax

```
call slaexc ( wantq, n, t, ldt, q, ldq, j1, n1, n2, work, info )
```

```
call dlaexc ( wantq, n, t, ldt, q, ldq, j1, n1, n2, work, info )
```

Description

This routine swaps adjacent diagonal blocks T_{11} and T_{22} of order 1 or 2 in an upper quasi-triangular matrix T by an orthogonal similarity transformation.
T must be in Schur canonical form, that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign.

Input Parameters

wantq	`LOGICAL`. If *wantq* =`.TRUE.`, accumulate the transformation in the matrix Q; If *wantq* =`.FALSE.`, do not accumulate the transformation.
n	`INTEGER`. The order of the matrix T ($n \geq 0$).
t, q	`REAL` for `slaexc` `DOUBLE PRECISION` for `dlaexc` Arrays: `t(ldt,*)` contains on entry the upper quasi-triangular matrix T, in Schur canonical form. The second dimension of `t` must be at least max(1, *n*). `q(ldq,*)` contains on entry, if *wantq* =`.TRUE.`, the orthogonal matrix Q. If *wantq* =`.FALSE.`, *q* is not referenced. The second dimension of *q* must be at least max(1, *n*).
ldt	`INTEGER`. The first dimension of `t`; at least max(1, *n*).
ldq	`INTEGER`. The first dimension of *q*; If *wantq* =`.FALSE.`, then *ldq* ≥ 1. If *wantq* =`.TRUE.`, then *ldq* ≥ max(1,*n*).
j1	`INTEGER`. The index of the first row of the first block T_{11}.
n1	`INTEGER`. The order of the first block T_{11} (*n1* = 0, 1, or 2).
n2	`INTEGER`. The order of the second block T_{22} (*n2* = 0, 1, or 2).

work REAL for slaexc;
DOUBLE PRECISION for dlaexc.
Workspace array, DIMENSION (*n*).

Output Parameters

t On exit, the updated matrix *T*, again in Schur canonical form.

q On exit, if *wantq* =.TRUE., the updated matrix *Q*.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = 1, the transformed matrix *T* would be too far from Schur form; the blocks are not swapped and *T* and *Q* are unchanged.

?lag2

Computes the eigenvalues of a 2-by-2 generalized eigenvalue problem, with scaling as necessary to avoid over-/underflow.

Syntax

```
call slag2 ( a, lda, b, ldb, safmin, scale1, scale2, wr1, wr2, wi )
call dlag2 ( a, lda, b, ldb, safmin, scale1, scale2, wr1, wr2, wi )
```

Description

This routine computes the eigenvalues of a 2 x 2 generalized eigenvalue problem $A - wB$, with scaling as necessary to avoid over-/underflow. The scaling factor, s, results in a modified eigenvalue equation

$$sA - wB,$$

where s is a non-negative scaling factor chosen so that w, wB, and sA do not overflow and, if possible, do not underflow, either.

Input Parameters

a, b REAL for slag2
DOUBLE PRECISION for dlag2
Arrays:

a(*lda*,2) contains, on entry, the 2 x 2 matrix *A*. It is assumed that its 1-norm is less than 1/*safmin*. Entries less than sqrt(*safmin*)*norm(*A*) are subject to being treated as zero.

b(*ldb*,2) contains, on entry, the 2 x 2 upper triangular matrix *B*. It is assumed that the one-norm of *B* is less than 1/*safmin*. The diagonals should be at least sqrt(*safmin*) times the largest element of *B* (in absolute value); if a diagonal is smaller than that, then +/- sqrt(*safmin*) will be used instead of that diagonal.

lda — INTEGER. The first dimension of *a*; *lda* ≥ 2.

ldb — INTEGER. The first dimension of *b*; *ldb* ≥ 2.

safmin — REAL for slag2;
DOUBLE PRECISION for dlag2.
The smallest positive number such that 1/*safmin* does not overflow. (This should always be ?lamch('S') - it is an argument in order to avoid having to call ?lamch frequently.)

Output Parameters

scale1 — REAL for slag2;
DOUBLE PRECISION for dlag2.
A scaling factor used to avoid over-/underflow in the eigenvalue equation which defines the first eigenvalue. If the eigenvalues are complex, then the eigenvalues are (*wr1* +/- *wi* ***i***)/*scale1* (which may lie outside the exponent range of the machine), *scale1*=*scale2*, and *scale1* will always be positive. If the eigenvalues are real, then the first (real) eigenvalue is *wr1* / *scale1* , but this may overflow or underflow, and in fact, *scale1* may be zero or less than the underflow threshhold if the exact eigenvalue is sufficiently large.

scale2 — REAL for slag2;
DOUBLE PRECISION for dlag2.
A scaling factor used to avoid over-/underflow in the eigenvalue equation which defines the second eigenvalue. If the eigenvalues are complex, then *scale2*=*scale1*. If the eigenvalues are real, then the second (real) eigenvalue is *wr2* / *scale2* , but this may overflow or underflow, and in fact, *scale2* may be zero or less than the underflow threshold if the exact eigenvalue is sufficiently large.

wr1
: `REAL` for `slag2`;
`DOUBLE PRECISION` for `dlag2`.
If the eigenvalue is real, then *wr1* is *scale1* times the eigenvalue closest to the (2,2) element of AB^{-1}. If the eigenvalue is complex, then *wr1*=*wr2* is *scale1* times the real part of the eigenvalues.

wr2
: `REAL` for `slag2`;
`DOUBLE PRECISION` for `dlag2`.
If the eigenvalue is real, then *wr2* is *scale2* times the other eigenvalue. If the eigenvalue is complex, then *wr1*=*wr2* is *scale1* times the real part of the eigenvalues.

wi
: `REAL` for `slag2`;
`DOUBLE PRECISION` for `dlag2`.
If the eigenvalue is real, then *wi* is zero. If the eigenvalue is complex, then *wi* is *scale1* times the imaginary part of the eigenvalues. *wi* will always be non-negative.

?lags2

Computes 2-by-2 orthogonal matrices U, V, and Q, and applies them to matrices A and B such that the rows of the transformed A and B are parallel.

Syntax

```
call slags2 ( upper, a1, a2, a3, b1, b2, b3, csu, snu,
      csv, snv, csq, snq  )
call dlags2 ( upper, a1, a2, a3, b1, b2, b3, csu, snu,
      csv, snv, csq, snq  )
```

Description

This routine computes 2-by-2 orthogonal matrices U, V and Q, such that if *upper* =`.TRUE.`, then

$$U'*A*Q = U'*\begin{bmatrix} A_1 & A_2 \\ 0 & A_3 \end{bmatrix}*Q = \begin{bmatrix} x & 0 \\ x & x \end{bmatrix}$$

and

$$V'*B*Q = V'*\begin{bmatrix} B_1 & B_2 \\ 0 & B_3 \end{bmatrix}*Q = \begin{bmatrix} x & 0 \\ x & x \end{bmatrix}$$

or if *upper* =`.FALSE.` , then

$$U'*A*Q = U'*\begin{bmatrix} A_1 & 0 \\ A_2 & A_3 \end{bmatrix}*Q = \begin{bmatrix} x & x \\ 0 & x \end{bmatrix}$$

and

$$V'*B*Q = V'*\begin{bmatrix} B_1 & 0 \\ B_2 & B_3 \end{bmatrix}*Q = \begin{bmatrix} x & x \\ 0 & x \end{bmatrix}$$

The rows of the transformed *A* and *B* are parallel, where

$$U = \begin{bmatrix} csu & snu \\ -snu & csu \end{bmatrix}, \quad V = \begin{bmatrix} csv & snv \\ -snv & csv \end{bmatrix}, \quad Q = \begin{bmatrix} csq & snq \\ -snq & csq \end{bmatrix}$$

Here Z' denotes the transpose of Z.

Input Parameters

upper — `LOGICAL`.
If *upper* =`.TRUE.`, the input matrices *A* and *B* are upper triangular;
If *upper* =`.FALSE.`, the input matrices *A* and *B* are lower triangular.

a1, a2, a3 — `REAL` for `slags2`
`DOUBLE PRECISION` for `dlags2`
On entry, *a1, a2* and *a3* are elements of the input 2-by-2 upper (lower) triangular matrix *A*.

b1, b2, b3 REAL for slags2
DOUBLE PRECISION for dlags2
On entry, *b1, b2* and *b3* are elements of the input 2-by-2 upper (lower) triangular matrix *B*.

Output Parameters

csu, snu REAL for slags2
DOUBLE PRECISION for dlags2
The desired orthogonal matrix *U*.

csv, snv REAL for slags2
DOUBLE PRECISION for dlags2
The desired orthogonal matrix *V*.

csq, snq REAL for slags2
DOUBLE PRECISION for dlags2
The desired orthogonal matrix *Q*.

?lagtf

Computes an LU factorization of a matrix $T-\lambda I$, where T is a general tridiagonal matrix, and λ a scalar, using partial pivoting with row interchanges.

Syntax

```
call slagtf ( n, a, lambda, b, c, tol, d, in, info )
call dlagtf ( n, a, lambda, b, c, tol, d, in, info )
```

Description

This routine factorizes the matrix (T - *lambda***I*), where T is an *n*-by-*n* tridiagonal matrix and *lambda* is a scalar, as

T - *lambda**$I = P L U$,

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 implicit row scaling. The parameter *lambda* is included in the routine so that `?lagtf` may be used, in conjunction with `?lagts`, to obtain eigenvectors of *T* by inverse iteration..

Input Parameters

n — `INTEGER`. The order of the matrix *T* ($n \geq 0$).

a, b, c — `REAL` for `slagtf`
`DOUBLE PRECISION` for `dlagtf`
Arrays, dimension *a*(*n*) , *b*(*n*-1), *c*(*n*-1):
On entry, *a*(*) must contain the diagonal elements of the matrix *T*.
On entry, *b*(*) must contain the (*n*-1) super-diagonal elements of *T*.
On entry, *c*(*) must contain the (*n*-1) sub-diagonal elements of *T*.

tol — `REAL` for `slagtf`
`DOUBLE PRECISION` for `dlagtf`
On entry, a relative tolerance used to indicate whether or not the matrix (*T* - *lambda***I*) is nearly singular. *tol* should normally be chose 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*.

Output Parameters

a — On exit, *a* is overwritten by the *n* diagonal elements of the upper triangular matrix *U* of the factorization of *T*.

b — On exit, *b* is overwritten by the *n*-1 super-diagonal elements of the matrix *U* of the factorization of *T*.

c — On exit, *c* is overwritten by the *n*-1 sub-diagonal elements of the matrix *L* of the factorization of *T*.

d — `REAL` for `slagtf`
`DOUBLE PRECISION` for `dlagtf`
Array, dimension (*n*-2).
On exit, *d* is overwritten by the *n*-2 second super-diagonal elements of the matrix *U* of the factorization of *T*.

in — `INTEGER`.
Array, dimension (*n*).
On exit, *in* contains details of the permutation matrix *P*. If an interchange occurred at the `k`-th step of the elimination, then *in*(`k`) = 1, otherwise *in*(`k`) =

0. The element `in`(n) returns the smallest positive integer `j` such that

`abs(` u`(j,j) )` ≤ `norm( (`T `-` `lambda*`I`)(j) )*tol`,

where norm(A(`j`)) denotes the sum of the absolute values of the `j`-th row of the matrix A. If no such `j` exists then `in`(n) is returned as zero. If `in`(n) is returned as positive, then a diagonal element of U is small, indicating that (T - `lambda*`I) is singular or nearly singular.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = -*k*, the *k*th parameter had an illegal value.

?lagtm

Performs a matrix-matrix product of the form C = αAB+βC, where A is a tridiagonal matrix, B and C are rectangular matrices, and α and β are scalars, which may be 0, 1, or -1.

Syntax

```
call slagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb)
call dlagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb)
call clagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb)
call zlagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb)
```

Description

This routine performs a matrix-vector product of the form :

B := `alpha*`A`*`X + `beta*`B

where A is a tridiagonal matrix of order n, B and X are n-by-*nrhs* matrices, and *alpha* and *beta* are real scalars, each of which may be 0., 1., or -1.

Input Parameters

trans — `CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`.
Indicates the form of the equations:
If *trans* = `'N'`, then B := `alpha*`A`*`X + `beta*`B
(no transpose);

If *trans* = 'T', then $B := alpha*A^T*X + beta*B$ (transpose);
If *trans* = 'C', then $B := alpha*A^H*X + beta*B$ (conjugate transpose)

n — INTEGER. The order of the matrix A ($n \geq 0$).

nrhs — INTEGER. The number of right-hand sides, i.e., the number of columns in X and B ($nrhs \geq 0$).

alpha, beta — REAL for slagtm/clagtm
DOUBLE PRECISION for dlagtm/zlagtm
The scalars α and β. *alpha* must be 0., 1., or -1.; otherwise, it is assumed to be 0. *beta* must be 0., 1., or -1.; otherwise, it is assumed to be 1.

dl,d,du — REAL for slagtm
DOUBLE PRECISION for dlagtm
COMPLEX for clagtm
COMPLEX*16 for zlagtm.
Arrays: *dl*(*n* - 1), *d*(*n*), *du*(*n* - 1).
The array *dl* contains the (n - 1) sub-diagonal elements of T.
The array *d* contains the *n* diagonal elements of T.
The array *du* contains the (n - 1) super-diagonal elements of T.

x, b — REAL for slagtm
DOUBLE PRECISION for dlagtm
COMPLEX for clagtm
COMPLEX*16 for zlagtm.
Arrays:
x(*ldx*, *) contains the *n*-by-*nrhs* matrix X. The second dimension of *x* must be at least max(1, *nrhs*).

b(*ldb*, *) contains the *n*-by-*nrhs* matrix B. The second dimension of *b* must be at least max(1, *nrhs*).

ldx — INTEGER. The leading dimension of the array *x*;
$ldx \geq \max(1, n)$.

ldb — INTEGER. The leading dimension of the array *b*;
$ldb \geq \max(1, n)$.

Output Parameters

b — Overwritten by the matrix expression
$B := alpha*A*X + beta*B$

?lagts

Solves the system of equations $(T-\lambda I)x = y$ or $(T-\lambda I)^T x = y$,where T is a general tridiagonal matrix and λ a scalar, using the LU factorization computed by `?lagtf`.

Syntax

```
call slagts ( job, n, a, b, c, d, in, y, tol, info )
call dlagts ( job, n, a, b, c, d, in, y, tol, info )
```

Description

This routine may be used to solve for *x* one of the systems of equations:

$(T - lambda*I)*x = y$ or $(T - lambda*I)'*x = y$,
where *T* is an *n*-by-*n* tridiagonal matrix, following the factorization of $(T - lambda*I)$ as

$$T - lambda*I = P L U,$$

computed by the routine `?lagtf`.

The choice of equation to be solved is controlled by the argument *job*, and in each case there is an option to perturb zero or very small diagonal elements of *U*, this option being intended for use in applications such as inverse iteration.

Input Parameters

job `INTEGER`. Specifies the job to be performed by `?lagts` as follows:
= 1: The equations $(T - lambda*I)x = y$ are to be solved, but diagonal elements of *U* are not to be perturbed.

= -1: The equations $(T - lambda*I)x = y$ are to be solved and, if overflow would otherwise occur, the diagonal elements of *U* are to be perturbed. See argument *tol* below.

= 2: The equations $(T - lambda*I)'x = y$ are to be solved, but diagonal elements of *U* are not to be perturbed.

= -2: The equations $(T - \texttt{lambda}*I)'x = y$ are to be solved and, if overflow would otherwise occur, the diagonal elements of *U* are to be perturbed. See argument `tol` below.

n — `INTEGER`. The order of the matrix *T* ($n \geq 0$).

a, b, c, d — `REAL` for `slagts`
`DOUBLE PRECISION` for `dlagts`
Arrays, dimension *a*(*n*), *b*(*n*-1), *c*(*n*-1), *d*(*n*-2):
On entry, `a(*)` must contain the diagonal elements of *U* as returned from `?lagtf`.
On entry, `b(*)` must contain the first super-diagonal elements of *U* as returned from `?lagtf`.
On entry, `c(*)` must contain the sub-diagonal elements of *L* as returned from `?lagtf`.
On entry, `d(*)` must contain the second super-diagonal elements of *U* as returned from `?lagtf`.

in — `INTEGER`.
Array, dimension (*n*).
On entry, `in(*)` must contain details of the matrix *P* as returned from `?lagtf`.

y — `REAL` for `slagts`
`DOUBLE PRECISION` for `dlagts`
Array, dimension (*n*). On entry, the right hand side vector *y*.

tol — `REAL` for `slagtf`
`DOUBLE PRECISION` for `dlagtf`.
On entry, with *job* < 0, `tol` should be the minimum perturbation to be made to very small diagonal elements of *U*. `tol` should normally be chosen as about *eps**norm(*U*), where *eps* is the relative machine precision, but if `tol` is supplied as non-positive, then it is reset to *eps**max(abs(u(i,j))). If *job* > 0 then `tol` is not referenced.

Output Parameters

y — On exit, *y* is overwritten by the solution vector *x*.

tol — On exit, `tol` is changed as described in *Input Parameters* section above, only if `tol` is non-positive on entry. Otherwise `tol` is unchanged.

info — `INTEGER`.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

If `info` $= i > 0$, overflow would occur when computing the `i`th element of the solution vector x. This can only occur when `job` is supplied as positive and either means that a diagonal element of U is very small, or that the elements of the right-hand side vector y are very large.

?lagv2

Computes the Generalized Schur factorization of a real 2-by-2 matrix pencil (A,B) where B is upper triangular.

Syntax

```
call slagv2 ( a, lda, b, ldb, alphar, alphai, beta, csl,
      snl, csr, snr )
call dlagv2 ( a, lda, b, ldb, alphar, alphai, beta, csl,
      snl, csr, snr )
```

Description

This routine computes the Generalized Schur factorization of a real 2-by-2 matrix pencil (*A,B*) where *B* is upper triangular. The routine computes orthogonal (rotation) matrices given by `csl`, `snl` and `csr`, `snr` such that:

1) if the pencil (*A,B*) has two real eigenvalues (include 0/0 or 1/0 types), then

$$\begin{bmatrix} a_{11} & a_{12} \\ 0 & a_{22} \end{bmatrix} = \begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

$$\begin{bmatrix} b_{11} & b_{12} \\ 0 & b_{22} \end{bmatrix} = \begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ 0 & b_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

2) if the pencil (*A,B*) has a pair of complex conjugate eigenvalues, then

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

$$\begin{bmatrix} b_{11} & 0 \\ 0 & b_{22} \end{bmatrix} = \begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ 0 & b_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

where $b_{11} \geq b_{22} > 0$.

Input Parameters

a, b — `REAL` for `slagv2`
`DOUBLE PRECISION` for `dlagv2`
Arrays:
`a(lda,2)` contains the 2-by-2 matrix *A*;
`b(ldb,2)` contains the upper triangular 2-by-2 matrix *B*.

lda — `INTEGER`. The leading dimension of the array `a`;
$lda \geq 2$.

ldb — `INTEGER`. The leading dimension of the array `b`;
$ldb \geq 2$.

Output Parameters

a — On exit, `a` is overwritten by the "*A*-part" of the generalized Schur form.

b — On exit, `b` is overwritten by the "*B*-part" of the generalized Schur form.

alphar, alphai, beta — `REAL` for `slagv2`
`DOUBLE PRECISION` for `dlagv2`.
Arrays, dimension (2) each.

(`alphar`(k) + ***i*** * `alphai`(k))/`beta`(k) are the eigenvalues of the pencil (*A,B*), k=1,2 and ***i*** = sqrt(-1). Note that `beta`(k) may be zero.

csl, snl — `REAL` for `slagv2`
`DOUBLE PRECISION` for `dlagv2`
The cosine and sine of the left rotation matrix, respectively.

csr, snr — `REAL` for `slagv2`
`DOUBLE PRECISION` for `dlagv2`
The cosine and sine of the right rotation matrix, respectively.

?lahqr

Computes the eigenvalues and Schur factorization of an upper Hessenberg matrix, using the double-shift/single-shift QR algorithm.

Syntax

```
call slahqr ( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi,
      iloz, ihiz, z, ldz, info )
call dlahqr ( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi,
      iloz, ihiz, z, ldz, info )
call clahqr ( wantt, wantz, n, ilo, ihi, h, ldh, w,
      iloz, ihiz, z, ldz, info )
call zlahqr ( wantt, wantz, n, ilo, ihi, h, ldh, w,
      iloz, ihiz, z, ldz, info )
```

Description

This routine is an auxiliary routine called by `?hseqr` to update the eigenvalues and Schur decomposition already computed by `?hseqr`, by dealing with the Hessenberg submatrix in rows and columns *ilo* to *ihi*.

Input Parameters

wantt — `LOGICAL`.
If *wantt* = `.TRUE.`, the full Schur form *T* is required;
If *wantt* = `.FALSE.`, eigenvalues only are required.

wantz — `LOGICAL`.
If *wantz* = `.TRUE.`, the matrix of Schur vectors *Z* is required;
If *wantz* = `.FALSE.`, Schur vectors are not required.

n — `INTEGER`. The order of the matrix *H* ($n \geq 0$).

ilo, ihi — `INTEGER`.
It is assumed that *H* is already upper quasi-triangular in rows and columns *ihi*+1:*n*, and that *H*(*ilo*,*ilo*-1) = 0 (unless *ilo* = 1). The routine `?lahqr` works primarily with the Hessenberg submatrix in rows and columns *ilo* to *ihi*, but applies transformations to all of *H* if *wantt* = `.TRUE.`.
Constraints:
$1 \leq ilo \leq \max(1, ihi)$; $ihi \leq n$.

h, z	`REAL` for `slahqr` `DOUBLE PRECISION` for `dlahqr` `COMPLEX` for `clahqr` `COMPLEX*16` for `zlahqr`. Arrays: *h*(*ldh*,*) contains the upper Hessenberg matrix *H*. The second dimension of *h* must be at least max(1, *n*). *z*(*ldz*,*) If *wantz* =`.TRUE.`, then, on entry, *z* must contain the current matrix *Z* of transformations accumulated by `?hseqr`. If *wantz* =`.FALSE.`, then *z* is not referenced. The second dimension of *z* must be at least max(1, *n*) .
ldh	`INTEGER`. The first dimension of *h*; at least max(1, *n*).
ldz	`INTEGER`. The first dimension of *z*; at least max(1, *n*).
iloz, ihiz	`INTEGER`. Specify the rows of *Z* to which transformations must be applied if *wantz* =`.TRUE.`. $1 \le iloz \le ilo$; $ihi \le ihiz \le n$.

Output Parameters

h	On exit, if *wantt* =`.TRUE.`, *H* is upper quasi-triangular (upper triangular for complex flavors) in rows and columns *ilo*:*ihi*, with any 2-by-2 diagonal blocks in standard form. If *wantt* =`.FALSE.`, the contents of *H* are unspecified on exit.
wr, wi	`REAL` for `slahqr` `DOUBLE PRECISION` for `dlahqr` Arrays, `DIMENSION` at least max (1, *n*) each. Used with real flavors only. The real and imaginary parts, respectively, of the computed eigenvalues *ilo* to *ihi* are stored in the corresponding elements of *wr* and *wi*. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the i-th and (i+1)th, with *wi*(i) > 0 and *wi*(i+1) < 0. If *wantt* =`.TRUE.`, the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *H*, with *wr*(i) = *H*(i,i), and, if *H*(i:i+1, i:i+1) is a 2-by-2 diagonal block, *wi*(i) = sqrt(*H*(i+1,i)***H*(i,i+1)) and *wi*(i+1) = -*wi*(i).
w	`COMPLEX` for `clahqr` `COMPLEX*16` for `zlahqr`. Array, `DIMENSION` at least max (1, *n*). Used with complex flavors only.

The computed eigenvalues `ilo` to `ihi` are stored in the corresponding elements of `w`.
If `wantt` =`.TRUE.`, the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *H*, with `w`(i) = *H*(i,i).

`z` — If `wantz` =`.TRUE.`, then, on exit `z` has been updated; transformations are applied only to the submatrix *Z*(`iloz`:`ihiz`, `ilo`:`ihi`).

`info` — `INTEGER`.
If `info` = 0, the execution is successful.
If `info` = `i` > 0, `?lahqr` failed to compute all the eigenvalues `ilo` to `ihi` in a total of 30*(`ihi`-`ilo`+1) iterations; elements i+1:`ihi` of `wr` and `wi` (for `slahqr/dlahqr`) or `w` (for `clahqr/zlahqr`) contain those eigenvalues which have been successfully computed.

?lahrd

Reduces the first `nb` *columns of a general rectangular matrix A so that elements below the* `k`*-th subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A.*

Syntax

```
call slahrd ( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call dlahrd ( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call clahrd ( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call zlahrd ( n, k, nb, a, lda, tau, t, ldt, y, ldy )
```

Description

The routine reduces the first `nb` columns of a real/complex general `n`-by-(`n`-`k`+1) matrix *A* so that elements below the `k`-th subdiagonal are zero. The reduction is performed by an orthogonal/unitary similarity transformation $Q' A Q$. The routine returns the matrices *V* and *T* which determine *Q* as a block reflector $I - V T V'$, and also the matrix $Y = A V T$.

The matrix *Q* is represented as products of `nb` elementary reflectors:
$Q = H(1) H(2) \ldots H(nb)$

Each $H(i)$ has the form

$H(i) = I - tau*v*v'$

where *tau* is a real/complex scalar, and v is a real/complex vector.

This is an auxiliary routine called by `?gehrd`.

Input Parameters

n	`INTEGER`. The order of the matrix A ($n \geq 0$).
k	`INTEGER`. The offset for the reduction. Elements below the *k*-th subdiagonal in the first *nb* columns are reduced to zero.
nb	`INTEGER`. The number of columns to be reduced.
a	`REAL` for `slahrd` `DOUBLE PRECISION` for `dlahrd` `COMPLEX` for `clahrd` `COMPLEX*16` for `zlahrd`. Array `a(`*lda*, *n*-*k*+1`)` contains the *n*-by-(*n*-*k*+1) general matrix A to be reduced.
lda	`INTEGER`. The first dimension of *a*; at least max(1, *n*).
ldt	`INTEGER`. The first dimension of the output array *t*; must be at least max(1, *nb*).
ldy	`INTEGER`. The first dimension of the output array *y*; must be at least max(1, *n*).

Output Parameters

a	On exit, the elements on and above the *k*-th subdiagonal in the first *nb* columns are overwritten with the corresponding elements of the reduced matrix; the elements below the *k*-th subdiagonal, with the array *tau*, represent the matrix Q as a product of elementary reflectors. The other columns of *a* are unchanged. See *Application Notes* below.
tau	`REAL` for `slahrd` `DOUBLE PRECISION` for `dlahrd` `COMPLEX` for `clahrd` `COMPLEX*16` for `zlahrd`. Array, `DIMENSION` (*nb*). Contains scalar factors of the elementary reflectors.

`t, y` — `REAL` for `slahrd`
`DOUBLE PRECISION` for `dlahrd`
`COMPLEX` for `clahrd`
`COMPLEX*16` for `zlahrd`.

Arrays, dimension `t(ldt, nb)`, `y(ldy, nb)`.
The array `t` contains upper triangular matrix *T*.

The array `y` contains the *n*-by-*nb* matrix *Y* .

Application Notes

For the elementary reflector *H*(i) ,

v(1:i+*k*-1) = 0, *v*(i+*k*) = 1; *v*(i+*k*+1:*n*) is stored on exit in *a*(i+*k*+1:*n*, i) and *tau* is stored in `tau`(i).

The elements of the vectors *v* together form the (*n*-*k*+1)-by-*nb* matrix *V* which is needed, with *T* and *Y*, to apply the transformation to the unreduced part of the matrix, using an update of the form: $A := (I - V T V') * (A - Y V')$.
The contents of *A* on exit are illustrated by the following example with
n = 7, *k* = 3 and *nb* = 2:

$$\begin{bmatrix} a & h & a & a & a \\ a & h & a & a & a \\ a & h & a & a & a \\ h & h & a & a & a \\ v_1 & h & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

where *a* denotes an element of the original matrix *A*, *h* denotes a modified element of the upper Hessenberg matrix *H*, and v_i denotes an element of the vector defining *H*(i).

?laic1

Applies one step of incremental condition estimation.

Syntax

```
call slaic1 ( job, j, x, sest, w, gamma, sestpr, s, c )
call dlaic1 ( job, j, x, sest, w, gamma, sestpr, s, c )
call claic1 ( job, j, x, sest, w, gamma, sestpr, s, c )
call zlaic1 ( job, j, x, sest, w, gamma, sestpr, s, c )
```

Description

The routine `?laic1` applies one step of incremental condition estimation in its simplest version.

Let x, $\|x\|_2 = 1$ (where $\|a\|_2$ denotes the 2-norm of a), be an approximate singular vector of an `j`-by-`j` lower triangular matrix L, such that

$\|L*x\|_2 = sest$

Then `?laic1` computes `sestpr`, `s`, `c` such that the vector

$$xhat = \begin{bmatrix} s*x \\ c \end{bmatrix}$$

is an approximate singular vector of

$$Lhat = \begin{bmatrix} L & 0 \\ w' & gamma \end{bmatrix}$$

in the sense that

$\|Lhat *xhat\|_2 = sestpr$.

Depending on `job`, an estimate for the largest or smallest singular value is computed.

Note that $[s\ c]'$ and $sestpr^2$ is an eigenpair of the system (for `slaic1/claic`)

$$\operatorname{diag}(sest*sest, 0) + [alpha\ \ gamma] \ * \begin{bmatrix} alpha \\ gamma \end{bmatrix}$$

where $alpha = x'*w$;

or of the system (for `claic1/zlaic`)

$$\operatorname{diag}(sest*sest, 0) + [alpha\ \ gamma] \ * \begin{bmatrix} \operatorname{conjg}(alpha) \\ \operatorname{conjg}(gamma) \end{bmatrix}$$

where $alpha = \operatorname{conjg}(x)'*w$.

Input Parameters

job	`INTEGER`. If *job* =1, an estimate for the largest singular value is computed; If *job* =2, an estimate for the smallest singular value is computed;
j	`INTEGER`. Length of *x* and *w*.
x, w	`REAL` for `slaic1` `DOUBLE PRECISION` for `dlaic1` `COMPLEX` for `claic1` `COMPLEX*16` for `zlaic1`. Arrays, dimension (*j*) each. Contain vectors *x* and *w* , respectively.
sest	`REAL` for `slaic1/claic1`; `DOUBLE PRECISION` for `dlaic1/zlaic1`. Estimated singular value of *j*-by-*j* matrix *L*.
gamma	`REAL` for `slaic1` `DOUBLE PRECISION` for `dlaic1` `COMPLEX` for `claic1` `COMPLEX*16` for `zlaic1`. The diagonal element *gamma*.

Output Parameters

sestpr	`REAL` for `slaic1/claic1`; `DOUBLE PRECISION` for `dlaic1/zlaic1`. Estimated singular value of (*j*+1)-by-(*j*+1) matrix *Lhat*.

s, c REAL for slaic1
DOUBLE PRECISION for dlaic1
COMPLEX for claic1
COMPLEX*16 for zlaic1.
Sine and cosine needed in forming *xhat*.

?laln2

Solves a 1-by-1 or 2-by-2 linear system of equations of the specified form.

Syntax

```
call slaln2( ltrans, na, nw, smin, ca, a, lda, d1, d2,
      b, ldb, wr, wi, x, ldx, scale, xnorm, info )
call dlaln2( ltrans, na, nw, smin, ca, a, lda, d1, d2,
      b, ldb, wr, wi, x, ldx, scale, xnorm, info )
```

Description

The routine solves a system of the form

$(ca\ A - w\ D)\ X = s\ B$ or $(ca\ A' - w\ D)\ X = s\ B$

with possible scaling (s) and perturbation of A (A' means A-transpose.)

A is an *na*-by-*na* real matrix, *ca* is a real scalar, D is an *na*-by-*na* real diagonal matrix, w is a real or complex value, and X and B are *na*-by-1 matrices: real if w is real, complex if w is complex. The parameter *na* may be 1 or 2.

If w is complex, X and B are represented as *na*-by-2 matrices, the first column of each being the real part and the second being the imaginary part.

The routine computes the scaling factor s (≤ 1) so chosen that X can be computed without overflow. X is further scaled if necessary to assure that norm($ca\ A - w\ D$)*norm(X) is less than overflow.

If both singular values of ($ca\ A - w\ D$) are less than *smin*, *smin* * I (where I stands for identity) will be used instead of ($ca\ A - w\ D$). If only one singular value is less than *smin*, one element of ($ca\ A - w\ D$) will be perturbed enough to make the smallest singular value roughly *smin*. If both singular values are at least *smin*, ($ca\ A - w\ D$) will not be perturbed.
In any case, the perturbation will be at most some small multiple of

max(*smin*, *ulp* * norm(*ca A* - *w D*)).
The singular values are computed by infinity-norm approximations, and thus will only be correct to a factor of 2 or so.

NOTE. All input quantities are assumed to be smaller than overflow by a reasonable factor (see *bignum*).

Input Parameters

trans — `LOGICAL`.
If *trans* =`.TRUE.`, *A*- transpose will be used.
If *trans* =`.FALSE.`, *A* will be used (not transposed.)

na — `INTEGER`. The size of the matrix *A*. May only be 1 or 2.

nw — `INTEGER`. This parameter must be 1 if *w* is real, and 2 if *w* is complex. May only be 1or 2.

smin — `REAL` for `slaln2`
`DOUBLE PRECISION` for `dlaln2`.
The desired lower bound on the singular values of *A*. This should be a safe distance away from underflow or overflow, for example, between (*underflow/machine_precision*) and (*machine_precision* * *overflow*). (See *bignum* and *ulp*).

ca — `REAL` for `slaln2`
`DOUBLE PRECISION` for `dlaln2`.
The coefficient by which *A* is multiplied.

a — `REAL` for `slaln2`
`DOUBLE PRECISION` for `dlaln2`.
Array, `DIMENSION` (*lda*,*na*). The *na*-by-*na* matrix *A*.

lda — `INTEGER`. The leading dimension of *a*. Must be at least *na*.

d1, d2 — `REAL` for `slaln2`
`DOUBLE PRECISION` for `dlaln2`.
The (1,1) and (2,2) elements in the diagonal matrix *D,* respectively. *d2* is not used if *nw* = 1.

b	`REAL` for `slaln2` `DOUBLE PRECISION` for `dlaln2`. Array, `DIMENSION` (*ldb*,*nw*). The *na*-by-*nw* matrix *B* (right-hand side). If *nw* =2 (*w* is complex), column 1 contains the real part of *B* and column 2 contains the imaginary part.
ldb	`INTEGER`. The leading dimension of *b*. Must be at least *na*.
wr, wi	`REAL` for `slaln2` `DOUBLE PRECISION` for `dlaln2`. The real and imaginary part of the scalar *w*, respectively. *wi* is not used if *nw* = 1.
ldx	`INTEGER`. The leading dimension of the output array *x*. Must be at least *na*.

Output Parameters

x	`REAL` for `slaln2` `DOUBLE PRECISION` for `dlaln2`. Array, `DIMENSION` (*ldx*,*nw*). The *na*-by-*nw* matrix *X* (unknowns), as computed by the routine. If *nw* = 2 (*w* is complex), on exit, column 1 will contain the real part of *X* and column 2 will contain the imaginary part.
scale	`REAL` for `slaln2` `DOUBLE PRECISION` for `dlaln2`. The scale factor that *B* must be multiplied by to insure that overflow does not occur when computing *X*. Thus (*ca A* - *w D*) *X* will be *scale***B*, not *B* (ignoring perturbations of *A*.) It will be at most 1.
xnorm	`REAL` for `slaln2` `DOUBLE PRECISION` for `dlaln2`. The infinity-norm of *X*, when *X* is regarded as an *na*-by-*nw* real matrix.
info	`INTEGER`. An error flag. It will be zero if no error occurs, a negative number if an argument is in error, or a positive number if (*ca A* - *w D*) had to be perturbed. The possible values are:

If *info* = 0: no error occurred, and (*ca* A - w D) did not have to be perturbed.
If *info* = 1: (*ca* A - w D) had to be perturbed to make its smallest (or only) singular value greater than *smin*.

NOTE. In the interests of speed, this routine does not check the inputs for errors.

?lals0

Applies back multiplying factors in solving the least squares problem using divide and conquer SVD approach. Used by `?gelsd`.

Syntax

```
call slals0( icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm,
      givptr, givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z,
      k, c, s, work, info )
call dlals0( icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm,
      givptr, givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z,
      k, c, s, work, info )
call clals0 ( icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm,
      givptr, givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z,
      k, c, s, rwork, info )
call zlals0 ( icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm,
      givptr, givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z,
      k, c, s, rwork, info )
```

Description

The routine applies back the multiplying factors of either the left or right singular vector matrix of a diagonal matrix appended by a row to the right hand side matrix B in solving the least squares problem using the divide-and-conquer SVD approach.

For the left singular vector matrix, three types of orthogonal matrices are involved:

(1L) Givens rotations: the number of such rotations is *givptr*; the pairs of columns/rows they were applied to are stored in *givcol*; and the *c*- and *s*-values of these rotations are stored in *givnum*.

(2L) Permutation. The (*nl*+1)-st row of *B* is to be moved to the first row, and for j=2:*n*, *perm*(j)-th row of *B* is to be moved to the j-th row.

(3L) The left singular vector matrix of the remaining matrix.

For the right singular vector matrix, four types of orthogonal matrices are involved:

(1R) The right singular vector matrix of the remaining matrix.

(2R) If *sqre* = 1, one extra Givens rotation to generate the right null space.

(3R) The inverse transformation of (2L).

(4R) The inverse transformation of (1L).

Input Parameters

icompq	INTEGER. Specifies whether singular vectors are to be computed in factored form: If *icompq* = 0: Left singular vector matrix. If *icompq* = 1: Right singular vector matrix.
nl	INTEGER. The row dimension of the upper block. *nl* ≥ 1.
nr	INTEGER. The row dimension of the lower block. *nr* ≥ 1.
sqre	INTEGER. If *sqre* = 0: the lower block is an *nr*-by-*nr* square matrix. If *sqre* = 1: the lower block is an *nr*-by-(*nr*+1) rectangular matrix. The bidiagonal matrix has row dimension *n* = *nl* + *nr* + 1, and column dimension *m* = *n* + *sqre*.
nrhs	INTEGER. The number of columns of *b* and *bx*. Must be at least 1.
b	REAL for slals0 DOUBLE PRECISION for dlals0 COMPLEX for clals0

	`COMPLEX*16` for `zlals0`. Array, `DIMENSION` (*ldb*, *nrhs*). Contains the right hand sides of the least squares problem in rows 1 through *m*.
ldb	`INTEGER`. The leading dimension of *b*. Must be at least max(1,max(*m*, *n*)).
bx	`REAL` for `slals0` `DOUBLE PRECISION` for `dlals0` `COMPLEX` for `clals0` `COMPLEX*16` for `zlals0`. Workspace array, `DIMENSION` (*ldbx*, *nrhs*).
ldbx	`INTEGER`. The leading dimension of *bx*.
perm	`INTEGER`. Array, `DIMENSION` (*n*). The permutations (from deflation and sorting) applied to the two blocks.
givptr	`INTEGER`. The number of Givens rotations which took place in this subproblem.
givcol	`INTEGER`. Array, `DIMENSION` (*ldgcol*, 2). Each pair of numbers indicates a pair of rows/columns involved in a Givens rotation.
ldgcol	`INTEGER`. The leading dimension of *givcol*, must be at least *n*.
givnum	`REAL` for `slals0 /clals0` `DOUBLE PRECISION` for `dlals0/zlals0` Array, `DIMENSION` (*ldgnum*, 2). Each number indicates the `c` or *s* value used in the corresponding Givens rotation.
ldgnum	`INTEGER`. The leading dimension of arrays *difr*, *poles* and *givnum*, must be at least *k*.
poles	`REAL` for `slals0 /clals0` `DOUBLE PRECISION` for `dlals0/zlals0` Array, `DIMENSION` (*ldgnum*, 2). On entry, *poles*(1:*k*, 1) contains the new singular values obtained from solving the secular equation, and *poles*(1:*k*, 2) is an array containing the poles in the secular equation.

difl	REAL for slals0 /clals0 DOUBLE PRECISION for dlals0/zlals0 Array, DIMENSION (*k*). On entry, *difl*(*i*) is the distance between *i*-th updated (undeflated) singular value and the *i*-th (undeflated) old singular value.
difr	REAL for slals0 /clals0 DOUBLE PRECISION for dlals0/zlals0 Array, DIMENSION (*ldgnum*, 2). On entry, *difr*(*i*, 1) contains the distances between *i*-th updated (undeflated) singular value and the *i*+*1*-th (undeflated) old singular value. And *difr*(*i*, 2) is the normalizing factor for the *i*-th right singular vector.
z	REAL for slals0 /clals0 DOUBLE PRECISION for dlals0/zlals0 Array, DIMENSION (*k*). Contains the components of the deflation-adjusted updating row vector.
k	INTEGER. Contains the dimension of the non-deflated matrix. This is the order of the related secular equation. $1 \le k \le n$.
c	REAL for slals0 /clals0 DOUBLE PRECISION for dlals0/zlals0 Contains garbage if *sqre* =0 and the *c* value of a Givens rotation related to the right null space if *sqre* = 1.
s	REAL for slals0 /clals0 DOUBLE PRECISION for dlals0/zlals0 Contains garbage if *sqre* =0 and the *s* value of a Givens rotation related to the right null space if *sqre* = 1.
work	REAL for slals0 DOUBLE PRECISION for dlals0 Workspace array, DIMENSION (*k*). Used with real flavors only.
rwork	REAL for clals0 DOUBLE PRECISION for zlals0 Workspace array, DIMENSION (*k**(1+*nrhs*) + 2**nrhs*). Used with complex flavors only.

Output Parameters

b	On exit, contains the solution *X* in rows 1 through *n*.

info INTEGER.
If *info* = 0: successful exit.
If *info* = *-i* < 0, the *i*-th argument had an illegal value.

?lalsa

Computes the SVD of the coefficient matrix in compact form. Used by `?gelsd`.

Syntax

```
call slalsa ( icompq, smlsiz, n, nrhs, b, ldb, bx, ldbx,
              u, ldu, vt, k, difl, difr, z, poles, givptr,
             givcol, ldgcol, perm, givnum, c, s, work,
              iwork, info )
call dlalsa ( icompq, smlsiz, n, nrhs, b, ldb, bx, ldbx,
              u, ldu, vt, k, difl, difr, z, poles, givptr,
             givcol, ldgcol, perm, givnum, c, s, work,
              iwork, info )
call clalsa ( icompq, smlsiz, n, nrhs, b, ldb, bx, ldbx,
              u, ldu, vt, k, difl, difr, z, poles, givptr,
             givcol, ldgcol, perm, givnum, c, s, rwork,
              iwork, info )
call zlalsa ( icompq, smlsiz, n, nrhs, b, ldb, bx, ldbx,
              u, ldu, vt, k, difl, difr, z, poles, givptr,
             givcol, ldgcol, perm, givnum, c, s, rwork,
              iwork, info )
```

Description

The routine is an itermediate step in solving the least squares problem by computing the SVD of the coefficient matrix in compact form. The singular vectors are computed as products of simple orthorgonal matrices.

If *icompq* = 0, `?lalsa` applies the inverse of the left singular vector matrix of an upper bidiagonal matrix to the right hand side; and if
icompq = 1, the routine applies the right singular vector matrix to the right hand side. The singular vector matrices were generated in the compact form by `?lalsa`.

Input Parameters

icompq	`INTEGER`. Specifies whether the left or the right singular vector matrix is involved. If *icompq* = 0: left singular vector matrix is used If *icompq* = 1: right singular vector matrix is used.
smlsiz	`INTEGER`. The maximum size of the subproblems at the bottom of the computation tree.
n	`INTEGER`. The row and column dimensions of the upper bidiagonal matrix.
nrhs	`INTEGER`. The number of columns of *b* and *bx*. Must be at least 1.
b	`REAL` for `slalsa` `DOUBLE PRECISION` for `dlalsa` `COMPLEX` for `clalsa` `COMPLEX*16` for `zlalsa` Array, `DIMENSION` (*ldb*, *nrhs*). Contains the right hand sides of the least squares problem in rows 1 through *m*.
ldb	`INTEGER`. The leading dimension of *b* in the calling subprogram. Must be at least max(1,max(*m*, *n*)).
ldbx	`INTEGER`. The leading dimension of the output array *bx*.
u	`REAL` for `slalsa/clalsa` `DOUBLE PRECISION` for `dlalsa/zlalsa` Array, `DIMENSION` (*ldu*, *smlsiz*). On entry, *u* contains the left singular vector matrices of all subproblems at the bottom level.
ldu	`INTEGER`, *ldu* ≥ *n*. The leading dimension of arrays *u*, *vt*, *difl*, *difr*, *poles*, *givnum*, and *z*.
vt	`REAL` for `slalsa/clalsa` `DOUBLE PRECISION` for `dlalsa/zlalsa` Array, `DIMENSION` (*ldu*, *smlsiz* +1). On entry, contains the right singular vector matrices of all subproblems at the bottom level.
k	`INTEGER` array, `DIMENSION` (*n*).
difl	`REAL` for `slalsa/clalsa` `DOUBLE PRECISION` for `dlalsa/zlalsa` Array, `DIMENSION` (*ldu*, *nlvl*), where *nlvl* = $\mathrm{int}(\log_2(n/(smlsiz+1))) + 1$.

difr REAL for slalsa/clalsa
DOUBLE PRECISION for dlalsa/zlalsa
Array, DIMENSION (*ldu*, 2**nlvl*). On entry,
difl(*, *i*) and *difr*(*, 2*i* -1) record distances between singular values on the *i*-th level and singular values on the (*i* -1)-th level, and *difr*(*, 2*i*) record the normalizing factors of the right singular vectors matrices of subproblems on *i*-th level.

z REAL for slalsa/clalsa
DOUBLE PRECISION for dlalsa/zlalsa
Array, DIMENSION (*ldu*, *nlvl*). On entry, *z*(1, *i*) contains the components of the deflation- adjusted updating the row vector for subproblems on the *i*-th level.

poles REAL for slalsa/clalsa
DOUBLE PRECISION for dlalsa/zlalsa
Array, DIMENSION (*ldu*, 2**nlvl*).
On entry, *poles*(*, 2*i*-1: 2*i*) contains the new and old singular values involved in the secular equations on the *i*-th level.

givptr INTEGER.
Array, DIMENSION (*n*).
On entry, *givptr*(*i*) records the number of Givens rotations performed on the *i*-th problem on the computation tree.

givcol INTEGER.
Array, DIMENSION (*ldgcol*, 2**nlvl*). On entry, for each *i*, *givcol*(*, 2*i*-1: 2*i*) records the locations of Givens rotations performed on the *i*-th level on the computation tree.

ldgcol INTEGER, *ldgcol* ≥ *n*. The leading dimension of arrays *givcol* and *perm*.

perm INTEGER.
Array, DIMENSION (*ldgcol*, *nlvl*). On entry, *perm*(*, *i*) records permutations done on the *i*-th level of the computation tree.

givnum REAL for slalsa/clalsa
DOUBLE PRECISION for dlalsa/zlalsa
Array, DIMENSION (*ldu*, 2**nlvl*). On entry, *givnum*(*, 2*i*-1 : 2*i*) records the *c* and *s* values of Givens rotations performed on the *i*-th level on the computation tree.

c	`REAL` for `slalsa/clalsa` `DOUBLE PRECISION` for `dlalsa/zlalsa` Array, `DIMENSION` (*n*). On entry, if the *i*-th subproblem is not square, *c*(*i*) contains the *c* value of a Givens rotation related to the right null space of the *i*-th subproblem.
s	`REAL` for `slalsa/clalsa` `DOUBLE PRECISION` for `dlalsa/zlalsa` Array, `DIMENSION` (*n*). On entry, if the *i*-th subproblem is not square, *s*(*i*) contains the *s*-value of a Givens rotation related to the right null space of the *i*-th subproblem.
work	`REAL` for `slalsa` `DOUBLE PRECISION` for `dlalsa` Workspace array, `DIMENSION` at least (*n*). Used with real flavors only.
rwork	`REAL` for `clalsa` `DOUBLE PRECISION` for `zlalsa` Workspace array, `DIMENSION` at least max(*n*, (*smlsz*+1)**nrhs**3). Used with complex flavors only.
iwork	`INTEGER`. Workspace array, `DIMENSION` at least (3*n*).

Output Parameters

b	On exit, contains the solution *X* in rows 1 through *n*.
bx	`REAL` for `slalsa` `DOUBLE PRECISION` for `dlalsa` `COMPLEX` for `clalsa` `COMPLEX*16` for `zlalsa` Array, `DIMENSION` (*ldbx*, *nrhs*). On exit, the result of applying the left or right singular vector matrix to *b*.
info	`INTEGER`. If *info* = 0: successful exit If *info* = -*i* < 0, the *i*-th argument had an illegal value.

?lalsd

Uses the singular value decomposition of A to solve the least squares problem.

Syntax

```
call slalsd ( uplo, smlsiz, n, nrhs, d, e, b, ldb,
              rcond, rank, work, iwork, info )
call dlalsd ( uplo, smlsiz, n, nrhs, d, e, b, ldb,
              rcond, rank, work, iwork, info )
call clalsd ( uplo, smlsiz, n, nrhs, d, e, b, ldb,
              rcond, rank, work, rwork, iwork, info )
call zlalsd ( uplo, smlsiz, n, nrhs, d, e, b, ldb,
              rcond, rank, work, rwork, iwork, info )
```

Description

The routine uses the singular value decomposition of A to solve the least squares problem of finding X to minimize the Euclidean norm of each column of $A X$-B, where A is *n*-by-*n* upper bidiagonal, and X and B are *n*-by-*nrhs*. The solution X overwrites B.

The singular values of A smaller than *rcond* times the largest singular value are treated as zero in solving the least squares problem; in this case a minimum norm solution is returned. The actual singular values are returned in *d* in ascending order.

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray XMP, Cray YMP, Cray C 90, or Cray 2.

It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

Input Parameters

uplo — `CHARACTER*1`.
If *uplo* = 'U', *d* and *e* define an upper bidiagonal matrix.
If *uplo* = 'L', *d* and *e* define a lower bidiagonal matrix.

smlsiz — `INTEGER`. The maximum size of the subproblems at the bottom of the computation tree.

n — `INTEGER`. The dimension of the bidiagonal matrix. $n \geq 0$.

nrhs — `INTEGER`. The number of columns of *B*. Must be at least 1.

d — `REAL` for `slalsd/clalsd`
`DOUBLE PRECISION` for `dlalsd/zlalsd`
Array, `DIMENSION` (*n*). On entry, *d* contains the main diagonal of the bidiagonal matrix.

e — `REAL` for `slalsd/clalsd`
`DOUBLE PRECISION` for `dlalsd/zlalsd`
Array, `DIMENSION` (*n*-1). Contains the super-diagonal entries of the bidiagonal matrix. On exit, *e* is destroyed.

b — `REAL` for `slalsd`
`DOUBLE PRECISION` for `dlalsd`
`COMPLEX` for `clalsd`
`COMPLEX*16` for `zlalsd`
Array, `DIMENSION` (*ldb*,*nrhs*). On input, *b* contains the right hand sides of the least squares problem. On output, *b* contains the solution *X*.

ldb — `INTEGER`. The leading dimension of *b* in the calling subprogram. Must be at least max(1,*n*).

rcond — `REAL` for `slalsd/clalsd`
`DOUBLE PRECISION` for `dlalsd/zlalsd`
The singular values of *A* less than or equal to *rcond* times the largest singular value are treated as zero in solving the least squares problem.
If *rcond* is negative, machine precision is used instead.
For example, if diag(*S*)**X*=*B* were the least squares problem, where diag(*S*) is a diagonal matrix of singular values, the solution would be $X(i) = B(i) / S(i)$ if $S(i)$ is greater than *rcond* *max(*S*), and $X(i) = 0$ if $S(i)$ is less than or equal to *rcond* *max(*S*).

rank — `INTEGER`. The number of singular values of *A* greater than *rcond* times the largest singular value.

work — `REAL` for `slalsd`
`DOUBLE PRECISION` for `dlalsd`
`COMPLEX` for `clalsd`
`COMPLEX*16` for `zlalsd`
Workspace array.
`DIMENSION` for real flavors at least

$(9n+2n*smlsiz+8n*nlvl+n*nrhs+(smlsiz+1)^2)$,
where
$nlvl = \max(0, \mathrm{int}(\log_2(n / (smlsiz+1))) + 1)$.

DIMENSION for complex flavors at least ($n*nrhs$).

rwork — REAL for clalsd
DOUBLE PRECISION for zlalsd
Workspace array, used with complex flavors only. DIMENSION at least $(9n + 2n*smlsiz + 8n*nlvl + 3*mlsiz*nrhs + (smlsiz+1)^2)$,
where
$nlvl = \max(0, \mathrm{int}(\log_2(\min(m,n)/(smlsiz+1))) + 1)$.

iwork — INTEGER.
Workspace array, DIMENSION at least ($3n*nlvl + 11n$).

Output Parameters

d — On exit, if *info* = 0, *d* contains singular values of the bidiagonal matrix.

b — On exit, *b* contains the solution *X*.

info — INTEGER.
If *info* = 0: successful exit.
If *info* = -*i* < 0, the *i*-th argument had an illegal value.
If *info* > 0: The algorithm failed to compute a singular value while working on the submatrix lying in rows and columns *info*/(*n*+1) through mod(*info*,*n*+1).

?lamrg

Creates a permutation list to merge the entries of two independently sorted sets into a single set sorted in acsending order.

Syntax

```
call slamrg ( n1, n2, a, strd1, strd2, index )
call dlamrg ( n1, n2, a, strd1, strd2, index )
```

Description

The routine creates a permutation list which will merge the elements of *a* (which is composed of two independently sorted sets) into a single set which is sorted in ascending order.

Input Parameters

n1, *n2* — `INTEGER`.
These arguments contain the respective lengths of the two sorted lists to be merged.

a — `REAL` for `slamrg`
`DOUBLE PRECISION` for `dlamrg`.
Array, `DIMENSION` (*n1*+*n2*).
The first *n1* elements of *a* contain a list of numbers which are sorted in either ascending or descending order. Likewise for the final *n2* elements.

strd1, *strd2* — `INTEGER`.
These are the strides to be taken through the array *a*. Allowable strides are 1 and -1. They indicate whether a subset of *a* is sorted in ascending (*strdx* = 1) or descending (*strdx* = -1) order.

Output Parameters

index — `INTEGER`.
Array, `DIMENSION` (*n1*+*n2*).
On exit, this array will contain a permutation such that if *b*(*i*) = *a*(*index*(*i*)) for *i*=1, *n1*+*n2*, then *b* will be sorted in ascending order.

?langb

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of general band matrix.

Syntax

```
val = slangb ( norm, n, kl, ku, ab, ldab, work )
val = dlangb ( norm, n, kl, ku, ab, ldab, work )
val = clangb ( norm, n, kl, ku, ab, ldab, work )
```

`val = zlangb ( norm, n, kl, ku, ab, ldab, work )`

Description

The function returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an *n*-by-*n* band matrix *A*, with *kl* sub-diagonals and *ku* super-diagonals.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(A), if *norm* = '1' or 'O' or 'o'

= normI(A), if *norm* = 'I' or 'i'

= normF(A), if *norm* = 'F', 'f', 'E' or 'e'

where `norm1` denotes the 1-norm of a matrix (maximum column sum), `normI` denotes the infinity norm of a matrix (maximum row sum) and `normF` denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a matrix norm.

Input Parameters

norm — `CHARACTER*1`. Specifies the value to be returned by the routine as described above.

n — `INTEGER`. The order of the matrix *A*.
n ≥ 0. When *n* = 0, `?langb` is set to zero.

kl — `INTEGER`. The number of sub-diagonals of the matrix *A*. *kl* ≥ 0.

ku — `INTEGER`. The number of super-diagonals of the matrix *A*. *ku* ≥ 0.

ab — `REAL` for `slangb`
`DOUBLE PRECISION` for `dlangb`
`COMPLEX` for `clangb`
`COMPLEX*16` for `zlangb`
Array, `DIMENSION` (*ldab*,*n*). The band matrix *A*, stored in rows 1 to *kl*+*ku*+1. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:
ab(*ku*+1+*i*-*j*,*j*) = *a*(*i*,*j*)
for max(1,*j*-*ku*) ≤ *i* ≤ min(*n*,*j*+*kl*).

ldab — `INTEGER`. The leading dimension of the array *ab*.
ldab ≥ *kl*+*ku*+1.

work — REAL for slangb/clangb
DOUBLE PRECISION for dlangb/zlangb
Workspace array, DIMENSION (*lwork*), where
lwork ≥ *n* when *norm* = 'I'; otherwise, *work* is not referenced.

Output Parameters

val — REAL for slangb/clangb
DOUBLE PRECISION for dlangb/zlangb
Value returned by the function.

?lange

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general rectangular matrix.

Syntax

```
val = slange ( norm, m, n, a, lda, work )
val = dlange ( norm, m, n, a, lda, work )
val = clange ( norm, m, n, a, lda, work )
val = zlange ( norm, m, n, a, lda, work )
```

Description

The function ?lange returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex matrix A.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(A), if *norm* = '1' or 'O' or 'o'

= normI(A), if *norm* = 'I' or 'i'

= normF(A), if *norm* = 'F', 'f', 'E' or 'e' where `norm1` denotes the 1-norm of a matrix (maximum column sum), `normI` denotes the infinity norm of a matrix (maximum row sum) and `normF` denotes the Frobenius norm of a matrix (square root of sum of squares). Note that $\max(\text{abs}(A_{ij}))$ is not a matrix norm.

Input Parameters

norm — `CHARACTER*1`. Specifies the value to be returned in `?lange` as described above.

m — `INTEGER`. The number of rows of the matrix *A*.
$m \geq 0$. When $m = 0$, `?lange` is set to zero.

n — `INTEGER`. The number of columns of the matrix *A*.
$n \geq 0$. When $n = 0$, `?lange` is set to zero.

a — `REAL` for `slange`
`DOUBLE PRECISION` for `dlange`
`COMPLEX` for `clange`
`COMPLEX*16` for `zlange`
Array, `DIMENSION` (*lda*,*n*). The *m*-by-*n* matrix *A*.

lda — `INTEGER`. The leading dimension of the array *a*.
$lda \geq \max(m,1)$.

work — `REAL` for `slange` and `clange`.
`DOUBLE PRECISION` for `dlange` and `zlange`.
Workspace array, `DIMENSION` (*lwork*), where $lwork \geq m$ when *norm* = 'I'; otherwise, *work* is not referenced.

Output Parameters

val — `REAL` for `slange/clange`
`DOUBLE PRECISION` for `dlange/zlange`
Value returned by the function.

?langt

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general tridiagonal matrix.

Syntax

```
val = slangt ( norm, n, dl, d, du )
val = dlangt ( norm, n, dl, d, du )
val = clangt ( norm, n, dl, d, du )
val = zlangt ( norm, n, dl, d, du )
```

Description

The routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex tridiagonal matrix A.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(A), if *norm* = '1' or 'O' or 'o'

= normI(A), if *norm* = 'I' or 'i'

= normF(A), if *norm* = 'F', 'f', 'E' or 'e'

where norm1 denotes the 1-norm of a matrix (maximum column sum), normI denotes the infinity norm of a matrix (maximum row sum) and normF denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a matrix norm.

Input Parameters

norm	CHARACTER*1. Specifies the value to be returned in ?langt as described above.
n	INTEGER. The order of the matrix A. $n \geq 0$. When $n = 0$, ?langt is set to zero.
dl, d, du	REAL for slangt DOUBLE PRECISION for dlangt COMPLEX for clangt

COMPLEX*16 for zlangt
Arrays: *dl* (*n*-1), *d* (*n*), *du* (*n*-1).
The array *dl* contains the (*n*-1) sub-diagonal elements of *A*.
The array *d* contains the diagonal elements of *A*.
The array *du* contains the (*n*-1) super-diagonal elements of *A*.

Output Parameters

val REAL for slangt/clangt
DOUBLE PRECISION for dlangt/zlangt
Value returned by the function.

?lanhs

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of an upper Hessenberg matrix.

Syntax

```
val = slanhs ( norm, n, a, lda, work )
val = dlanhs ( norm, n, a, lda, work )
val = clanhs ( norm, n, a, lda, work )
val = zlanhs ( norm, n, a, lda, work )
```

Description

The function ?lanhs returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hessenberg matrix *A*.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(*A*), if *norm* = '1' or 'O' or 'o'

= normI(*A*), if *norm* = 'I' or 'i'

= normF(*A*), if *norm* = 'F', 'f', 'E' or 'e' where `norm1` denotes the 1-norm of a matrix (maximum column sum), `normI` denotes the infinity norm of a matrix (maximum row sum) and `normF` denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a matrix norm.

Input Parameters

norm	`CHARACTER*1`. Specifies the value to be returned in `?lanhs` as described above.
n	`INTEGER`. The order of the matrix *A*. *n* ≥ 0. When *n* = 0, `?lanhs` is set to zero.
a	`REAL` for `slanhs` `DOUBLE PRECISION` for `dlanhs` `COMPLEX` for `clanhs` `COMPLEX*16` for `zlanhs` Array, `DIMENSION` (*lda*,*n*). The *n*-by-*n* upper Hessenberg matrix *A*; the part of *A* below the first sub-diagonal is not referenced.
lda	`INTEGER`. The leading dimension of the array *a*. *lda* ≥ max(*n*,1).
work	`REAL` for `slanhs` and `clanhs`. `DOUBLE PRECISION` for `dlange` and `zlange`. Workspace array, `DIMENSION` (*lwork*), where *lwork* ≥ *n* when *norm* = '`I`'; otherwise, *work* is not referenced.

Output Parameters

val	`REAL` for `slanhs/clanhs` `DOUBLE PRECISION` for `dlanhs/zlanhs` Value returned by the function.

?lansb

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric band matrix.

Syntax

```
val = slansb ( norm, uplo, n, k, ab, ldab, work )
```

```
val = dlansb ( norm, uplo, n, k, ab, ldab, work )
val = clansb ( norm, uplo, n, k, ab, ldab, work )
val = zlansb ( norm, uplo, n, k, ab, ldab, work )
```

Description

The function `?lansb` returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an *n*-by-*n* real/complex symmetric band matrix *A*, with *k* super-diagonals.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(*A*), if *norm* = '1' or 'O' or 'o'

= normI(*A*), if *norm* = 'I' or 'i'

= normF(*A*), if *norm* = 'F', 'f', 'E' or 'e'

where `norm1` denotes the 1-norm of a matrix (maximum column sum), `normI` denotes the infinity norm of a matrix (maximum row sum) and `normF` denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a matrix norm.

Input Parameters

norm — `CHARACTER*1`. Specifies the value to be returned in `?lansb` as described above.

uplo — `CHARACTER*1`. Specifies whether the upper or lower triangular part of the band matrix *A* is supplied.
If *uplo* = 'U': upper triangular part is supplied;
If *uplo* = 'L': lower triangular part is supplied.

n — `INTEGER`. The order of the matrix *A*. $n \geq 0$.
When *n* = 0, `?lansb` is set to zero.

k — `INTEGER`. The number of super-diagonals or sub-diagonals of the band matrix *A*. $k \geq 0$.

ab — `REAL` for `slansb`
`DOUBLE PRECISION` for `dlansb`
`COMPLEX` for `clansb`
`COMPLEX*16` for `zlansb`
Array, `DIMENSION` (*ldab*,*n*). The upper or lower triangle of the symmetric

	band matrix A, stored in the first k+1 rows of *ab*. The *j*-th column of A is stored in the *j*-th column of the array *ab* as follows: if *uplo* = 'U', *ab*(k+1+*i*-*j*,*j*) = *a*(*i*,*j*) for max(1,*j*-*k*) ≤ *i* ≤ *j*; if *uplo* = 'L', *ab*(1+*i*-*j*,*j*) = *a*(*i*,*j*) for *j*≤*i*≤min(*n*,*j*+*k*).
ldab	INTEGER. The leading dimension of the array *ab*. *ldab* ≥ *k*+1.
work	REAL for slansb and clansb. DOUBLE PRECISION for dlansb and zlansb. Workspace array, DIMENSION (*lwork*), where *lwork* ≥ *n* when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val	REAL for slansb/clansb DOUBLE PRECISION for dlansb/zlansb Value returned by the function.

?lanhb

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hermitian band matrix.

Syntax

```
val = clanhb ( norm, uplo, n, k, ab, ldab, work )
val = zlanhb ( norm, uplo, n, k, ab, ldab, work )
```

Description

The routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an *n*-by-*n* Hermitian band matrix A, with *k* super-diagonals.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(A), if *norm* = '1' or 'O' or 'o'

= `normI`(A) ,   if *norm* = 'I' or 'i'

= `normF`(A) ,   if *norm* = 'F', 'f', 'E' or 'e'

where `norm1` denotes the 1-norm of a matrix (maximum column sum), `normI` denotes the infinity norm of a matrix (maximum row sum) and `normF` denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a matrix norm.

Input Parameters

norm	`CHARACTER*1`. Specifies the value to be returned in `?lanhb` as described above.
uplo	`CHARACTER*1`. Specifies whether the upper or lower triangular part of the band matrix A is supplied. If *uplo* = 'U': upper triangular part is supplied; If *uplo* = 'L': lower triangular part is supplied.
n	`INTEGER`. The order of the matrix A. $n \geq 0$. When $n = 0$, `?lanhb` is set to zero.
k	`INTEGER`. The number of super-diagonals or sub-diagonals of the band matrix A. $k \geq 0$.
ab	`COMPLEX` for `clanhb`. `COMPLEX*16` for `zlanhb`. Array, `DIMENSION` (*ldab*,*n*). The upper or lower triangle of the Hermitian band matrix A, stored in the first k+1 rows of *ab*. The *j*-th column of A is stored in the *j*-th column of the array *ab* as follows: if *uplo* = 'U', $ab(k+1+i-j,j) = a(i,j)$ for $\max(1,j-k) \leq i \leq j$; if *uplo* = 'L', $ab(1+i-j,j) = a(i,j)$ for $j \leq i \leq \min(n,j+k)$. Note that the imaginary parts of the diagonal elements need not be set and are assumed to be zero.
ldab	`INTEGER`. The leading dimension of the array *ab*. $ldab \geq k+1$.
work	`REAL` for `clanhb`. `DOUBLE PRECISION` for `zlanhb`. Workspace array, `DIMENSION` (*lwork*), where $lwork \geq n$ when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val — REAL for slanhb/clanhb
DOUBLE PRECISION for dlanhb/zlanhb
Value returned by the function.

?lansp

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric matrix supplied in packed form.

Syntax

```
val = slansp ( norm, uplo, n, ap, work )
val = dlansp ( norm, uplo, n, ap, work )
val = clansp ( norm, uplo, n, ap, work )
val = zlansp ( norm, uplo, n, ap, work )
```

Description

The function ?lansp returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix A, supplied in packed form.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(A) , if *norm* = '1' or 'O' or 'o'

= normI(A) , if *norm* = 'I' or 'i'

= normF(A) , if *norm* = 'F', 'f', 'E' or 'e'

where norm1 denotes the 1-norm of a matrix (maximum column sum), normI denotes the infinity norm of a matrix (maximum row sum) and normF denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a matrix norm.

Input Parameters

norm — CHARACTER*1. Specifies the value to be returned in ?lansp as described above.

uplo — CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix A is supplied.
If *uplo* = 'U': Upper triangular part of A is supplied
If *uplo* = 'L': Lower triangular part of A is supplied.

n — INTEGER. The order of the matrix A. $n \geq 0$. When $n = 0$, ?lansp is set to zero.

ap — REAL for slansp
DOUBLE PRECISION for dlansp
COMPLEX for clansp
COMPLEX*16 for zlansp
Array, DIMENSION ($n(n+1)/2$). The upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array *ap* as follows:
if *uplo* = 'U', $ap(i + (j-1)j/2) = A(i,j)$ for $1 \leq i \leq j$;
if *uplo* = 'L', $ap(i + (j-1)(2n-j)/2) = A(i,j)$ for $j \leq i \leq n$.

work — REAL for slansp and clansp.
DOUBLE PRECISION for dlansp and zlansp.
Workspace array, DIMENSION (*lwork*), where
$lwork \geq n$ when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val — REAL for slansp/clansp
DOUBLE PRECISION for dlansp/zlansp
Value returned by the function.

?lanhp

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix supplied in packed form.

Syntax

```
val = clanhp ( norm, uplo, n, ap, work )
val = zlanhp ( norm, uplo, n, ap, work )
```

Description

The function `?lanhp` returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix A, supplied in packed form.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(A), if *norm* = '1' or 'O' or 'o'

= normI(A), if *norm* = 'I' or 'i'

= normF(A), if *norm* = 'F', 'f', 'E' or 'e'

where `norm1` denotes the 1-norm of a matrix (maximum column sum), `normI` denotes the infinity norm of a matrix (maximum row sum) and `normF` denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a matrix norm.

Input Parameters

norm — `CHARACTER*1`. Specifies the value to be returned in `?lanhp` as described above.

uplo — `CHARACTER*1`. Specifies whether the upper or lower triangular part of the Hermitian matrix A is supplied.
If *uplo* = 'U': Upper triangular part of A is supplied
If *uplo* = 'L': Lower triangular part of A is supplied.

n	`INTEGER`. The order of the matrix *A*. *n* ≥ 0. When *n* = 0, `?lanhp` is set to zero.
ap	`COMPLEX` for `clanhp`. `COMPLEX*16` for `zlanhp`. Array, `DIMENSION` (*n*(*n*+1)/2). The upper or lower triangle of the Hermitian matrix *A*, packed columnwise in a linear array. The *j*-th column of *A* is stored in the array *ap* as follows: if *uplo* = 'U', *ap*(*i* + (*j*-1)*j*/2) = *A*(*i*,*j*) for 1≤ *i*≤ *j*; if *uplo* = 'L', *ap*(*i* + (*j*-1)(2*n*-*j*)/2) = *A*(*i*,*j*) for *j*≤*i*≤*n*.
work	`REAL` for `clanhp`. `DOUBLE PRECISION` for `zlanhp`. Workspace array, `DIMENSION` (*lwork*), where *lwork* ≥ *n* when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val	`REAL` for `clanhp`. `DOUBLE PRECISION` for `zlanhp`. Value returned by the function.

?lanst/?lanht

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real symmetric or complex Hermitian tridiagonal matrix.

Syntax

```
val = slanst ( norm, n, d, e )
val = dlanst ( norm, n, d, e )
val = clanht ( norm, n, d, e )
val = zlanht ( norm, n, d, e )
```

Description

The functions `?lanst/?lanht` return the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real symmetric or a complex Hermitian tridiagonal matrix A.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(A) , if *norm* = '1' or 'O' or 'o'

= normI(A) , if *norm* = 'I' or 'i'

= normF(A) , if *norm* = 'F', 'f', 'E' or 'e'

where `norm1` denotes the 1-norm of a matrix (maximum column sum), `normI` denotes the infinity norm of a matrix (maximum row sum) and `normF` denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a matrix norm.

Input Parameters

norm — `CHARACTER*1`. Specifies the value to be returned in `?lanst/?lanht` as described above.

n — `INTEGER`. The order of the matrix A.
$n \ge 0$. When $n = 0$, `?lanst/?lanht` is set to zero.

d — `REAL` for `slanst/clanht`
`DOUBLE PRECISION` for `dlanst/zlanht`
Array, `DIMENSION` (*n*). The diagonal elements of A.

e — `REAL` for `slanst`
`DOUBLE PRECISION` for `dlanst`
`COMPLEX` for `clanht`
`COMPLEX*16` for `zlanht`
Array, `DIMENSION` (*n*-1). The (*n*-1) sub-diagonal or super-diagonal elements of A.

Output Parameters

val — `REAL` for `slanst/clanht`
`DOUBLE PRECISION` for `dlanst/zlanht`
Value returned by the function.

?lansy

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix.

Syntax

```
val = slansy ( norm, uplo, n, a, lda, work )
val = dlansy ( norm, uplo, n, a, lda, work )
val = clansy ( norm, uplo, n, a, lda, work )
val = zlansy ( norm, uplo, n, a, lda, work )
```

Description

The function `?lansy` returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix A.

The value `val` returned by the function is:

`val` = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(A), if *norm* = '1' or 'O' or 'o'

= normI(A), if *norm* = 'I' or 'i'

= normF(A), if *norm* = 'F', 'f', 'E' or 'e'

where `norm1` denotes the 1-norm of a matrix (maximum column sum), `normI` denotes the infinity norm of a matrix (maximum row sum) and `normF` denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a matrix norm.

Input Parameters

norm — `CHARACTER*1`. Specifies the value to be returned in `?lansy` as described above.

uplo — `CHARACTER*1`. Specifies whether the upper or lower triangular part of the symmetric matrix A is to be referenced.
= 'U': Upper triangular part of A is referenced.
= 'L': Lower triangular part of A is referenced

n — `INTEGER`. The order of the matrix A. $n \geq 0$. When $n = 0$, `?lansy` is set to zero.

a — `REAL` for `slansy`
`DOUBLE PRECISION` for `dlansy`
`COMPLEX` for `clansy`
`COMPLEX*16` for `zlansy`
Array, `DIMENSION` (*lda*,*n*). The symmetric matrix A. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix A, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix A, and the strictly upper triangular part of *a* is not referenced.

lda — `INTEGER`. The leading dimension of the array *a*. $lda \geq \max(n,1)$.

work — `REAL` for `slansy` and `clansy`.
`DOUBLE PRECISION` for `dlansy` and `zlansy`.
Workspace array, `DIMENSION` (*lwork*), where $lwork \geq n$ when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val — `REAL` for `slansy/clansy`
`DOUBLE PRECISION` for `dlansy/zlansy`
Value returned by the function.

?lanhe

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix.

Syntax

```
val = clanhe ( norm, uplo, n, a, lda, work )
val = zlanhe ( norm, uplo, n, a, lda, work )
```

Description

The function `?lanhe` returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix A.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= `norm1`(A) , if *norm* = '1' or 'O' or 'o'

= `normI`(A) , if *norm* = 'I' or 'i'

= `normF`(A) , if *norm* = 'F', 'f', 'E' or 'e'

where `norm1` denotes the 1-norm of a matrix (maximum column sum), `normI` denotes the infinity norm of a matrix (maximum row sum) and `normF` denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a matrix norm.

Input Parameters

norm — `CHARACTER*1`. Specifies the value to be returned in `?lanhe` as described above.

uplo — `CHARACTER*1`. Specifies whether the upper or lower triangular part of the Hermitian matrix A is to be referenced.
= 'U': Upper triangular part of A is referenced.
= 'L': Lower triangular part of A is referenced

n — `INTEGER`. The order of the matrix A.
$n \geq 0$. When $n = 0$, `?lanhe` is set to zero.

a — `COMPLEX` for `clanhe`.
`COMPLEX*16` for `zlanhe`.
Array, `DIMENSION` (*lda*,*n*). The Hermitian matrix A.
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix A, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix A, and the strictly upper triangular part of *a* is not referenced.

lda — `INTEGER`. The leading dimension of the array *a*.
lda $\geq$ max(*n*,1).

work	`REAL` for `clanhe`. `DOUBLE PRECISION` for `zlanhe`. Workspace array, `DIMENSION` (*lwork*), where *lwork* ≥ *n* when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val	`REAL` for `clanhe`. `DOUBLE PRECISION` for `zlanhe`. Value returned by the function.

?lantb

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular band matrix.

Syntax

```
val = slantb ( norm, uplo, diag, n, k, ab, ldab, work )
val = dlantb ( norm, uplo, diag, n, k, ab, ldab, work )
val = clantb ( norm, uplo, diag, n, k, ab, ldab, work )
val = zlantb ( norm, uplo, diag, n, k, ab, ldab, work )
```

Description

The function `?lantb` returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an *n*-by-*n* triangular band matrix *A*, with (*k* + 1) diagonals.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(A) , if *norm* = '1' or 'O' or 'o'

= normI(A) , if *norm* = 'I' or 'i'

= normF(A) , if *norm* = 'F', 'f', 'E' or 'e' where `norm1` denotes the 1-norm of a matrix (maximum column sum), `normI` denotes the infinity norm of a matrix (maximum row sum) and `normF` denotes the Frobenius norm of a matrix (square root of sum of squares). Note that $\max(\text{abs}(A_{ij}))$ is not a matrix norm.

Input Parameters

norm — `CHARACTER*1`. Specifies the value to be returned in `?lantb` as described above.

uplo — `CHARACTER*1`. Specifies whether the matrix *A* is upper or lower triangular.
= 'U': Upper triangular
= 'L': Lower triangular.

diag — `CHARACTER*1`. Specifies whether or not the matrix *A* is unit triangular.
= 'N': Non-unit triangular
= 'U': Unit triangular.

n — `INTEGER`. The order of the matrix *A*.
$n \geq 0$. When $n = 0$, `?lantb` is set to zero.

k — `INTEGER`. The number of super-diagonals of the matrix *A* if *uplo* = 'U', or the number of sub-diagonals of the matrix *A* if *uplo* = 'L'. $k \geq 0$.

ab — `REAL` for `slantb`
`DOUBLE PRECISION` for `dlantb`
`COMPLEX` for `clantb`
`COMPLEX*16` for `zlantb`
Array, `DIMENSION` (*ldab*,*n*). The upper or lower triangular band matrix *A*, stored in the first k+1 rows of *ab*. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:
if *uplo* = 'U', $ab(k+1+i-j,j) = a(i,j)$ for
$\max(1,j-k) \leq i \leq j$;
if *uplo* = 'L', $ab(1+i-j,j) = a(i,j)$ for
$j \leq i \leq \min(n,j+k)$.
Note that when *diag* = 'U', the elements of the array *ab* corresponding to the diagonal elements of the matrix *A* are not referenced, but are assumed to be one.

ldab — `INTEGER`. The leading dimension of the array *ab*.
$ldab \geq k+1$.

work — `REAL` for `slantb` and `clantb`.
`DOUBLE PRECISION` for `dlantb` and `zlantb`.
Workspace array, `DIMENSION` (*lwork*), where
$lwork \geq n$ when *norm* = 'I' ; otherwise, *work* is not referenced.

Output Parameters

`val` — `REAL` for `slantb`/`clantb`.
`DOUBLE PRECISION` for `dlantb`/`zlantb`.
Value returned by the function.

?lantp

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular matrix supplied in packed form.

Syntax

```
val = slantp ( norm, uplo, diag, n, ap, work )
val = dlantp ( norm, uplo, diag, n, ap, work )
val = clantp ( norm, uplo, diag, n, ap, work )
val = zlantp ( norm, uplo, diag, n, ap, work )
```

Discussion

The function `?lantp` returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular matrix A, supplied in packed form.

The value `val` returned by the function is:

`val` = max(abs(A_{ij})), if *norm* = ‘`M`’ or ‘`m`’

= `norm1`(A), if *norm* = ‘`1`’ or ‘O’ or ‘`o`’

= `normI`(A), if *norm* = ‘`I`’ or ‘`i`’

= `normF`(A), if *norm* = ‘`F`’, ‘`f`’, ‘`E`’ or ‘`e`’

where `norm1` denotes the 1-norm of a matrix (maximum column sum), `normI` denotes the infinity norm of a matrix (maximum row sum) and `normF` denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a matrix norm.

Input Parameters

norm — `CHARACTER*1`. Specifies the value to be returned in `?lantp` as described above.

uplo — `CHARACTER*1`. Specifies whether the matrix *A* is upper or lower triangular.
= 'U': Upper triangular
= 'L': Lower triangular.

diag — `CHARACTER*1`. Specifies whether or not the matrix *A* is unit triangular.
= 'N': Non-unit triangular
= 'U': Unit triangular.

n — `INTEGER`. The order of the matrix *A*.
$n \geq 0$. When $n = 0$, `?lantp` is set to zero.

ap — `REAL` for `slantp`
`DOUBLE PRECISION` for `dlantp`
`COMPLEX` for `clantp`
`COMPLEX*16` for `zlantp`
Array, `DIMENSION` $(n(n+1)/2)$. The upper or lower triangular matrix *A*, packed columnwise in a linear array. The *j*-th column of *A* is stored in the array *ap* as follows:
if *uplo* = 'U', $AP(i + (j-1)j/2) = a(i,j)$
for $1 \leq i \leq j$;
if *uplo* = 'L', $ap(i + (j-1)(2n-j)/2) = a(i,j)$
for $j \leq i \leq n$.
Note that when *diag* = 'U', the elements of the array *ap* corresponding to the diagonal elements of the matrix *A* are not referenced, but are assumed to be one.

work — `REAL` for `slantp` and `clantp`.
`DOUBLE PRECISION` for `dlantp` and `zlantp`.
Workspace array, `DIMENSION` (*lwork*), where
$lwork \geq n$ when *norm* = 'I' ; otherwise, *work* is not referenced.

Output Parameters

val — `REAL` for `slantp/clantp`.
`DOUBLE PRECISION` for `dlantp/zlantp`.
Value returned by the function.

?lantr

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix.

Syntax

```
val = slantr ( norm, uplo, diag, m, n, a, lda, work )
val = dlantr ( norm, uplo, diag, m, n, a, lda, work )
val = clantr ( norm, uplo, diag, m, n, a, lda, work )
val = zlantr ( norm, uplo, diag, m, n, a, lda, work )
```

Description

The function `?lantr` returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix A.

The value *val* returned by the function is:

val = $\max(\text{abs}(A_{ij}))$, if *norm* = 'M' or 'm'

= norm1(A), if *norm* = '1' or 'O' or 'o'

= normI(A), if *norm* = 'I' or 'i'

= normF(A), if *norm* = 'F', 'f', 'E' or 'e'

where `norm1` denotes the 1-norm of a matrix (maximum column sum), `normI` denotes the infinity norm of a matrix (maximum row sum) and `normF` denotes the Frobenius norm of a matrix (square root of sum of squares). Note that $\max(\text{abs}(A_{ij}))$ is not a matrix norm.

Input Parameters

norm — `CHARACTER*1`. Specifies the value to be returned in `?lantr` as described above.

uplo — `CHARACTER*1`. Specifies whether the matrix A is upper or lower trapezoidal.
= 'U': Upper trapezoidal
= 'L': Lower trapezoidal.
Note that A is triangular instead of trapezoidal if $m = n$.

diag	`CHARACTER*1`. Specifies whether or not the matrix *A* has unit diagonal. = 'N': Non-unit diagonal = 'U': Unit diagonal.
m	`INTEGER`. The number of rows of the matrix *A*. $m \geq 0$, and if *uplo* = 'U', $m \leq n$. When *m* = 0, `?lantr` is set to zero.
n	`INTEGER`. The number of columns of the matrix *A*. $n \geq 0$, and if *uplo* = 'L', $n \leq m$. When *n* = 0, `?lantr` is set to zero.
a	`REAL` for `slantr` `DOUBLE PRECISION` for `dlantr` `COMPLEX` for `clantr` `COMPLEX*16` for `zlantr` Array, `DIMENSION` (*lda*,*n*). The trapezoidal matrix *A* (*A* is triangular if *m* = *n*). If *uplo* = 'U', the leading *m*-by-*n* upper trapezoidal part of the array *a* contains the upper trapezoidal matrix, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *m*-by-*n* lower trapezoidal part of the array *a* contains the lower trapezoidal matrix, and the strictly upper triangular part of *a* is not referenced. Note that when *diag* = 'U', the diagonal elements of *a* are not referenced and are assumed to be one.
lda	`INTEGER`. The leading dimension of the array *a*. *lda* $\geq$ max(*m*,1).
work	`REAL` for `slantr/clantrp`. `DOUBLE PRECISION` for `dlantr/zlantr`. Workspace array, `DIMENSION` (*lwork*), where *lwork* $\geq$ *m* when *norm* = 'I' ; otherwise, *work* is not referenced.

Output Parameters

val	`REAL` for `slantr/clantrp`. `DOUBLE PRECISION` for `dlantr/zlantr`. Value returned by the function.

?lanv2

Computes the Schur factorization of a real 2-by-2 nonsymmetric matrix in standard form.

Syntax

```
call slanv2 ( a, b, c, d, rt1r, rt1i, rt2r, rt2i, cs, sn )
call dlanv2 ( a, b, c, d, rt1r, rt1i, rt2r, rt2i, cs, sn )
```

Description

The routine computes the Schur factorization of a real 2-by-2 nonsymmetric matrix in standard form:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} cs & -sn \\ sn & cs \end{bmatrix} \begin{bmatrix} aa & bb \\ cc & dd \end{bmatrix} \begin{bmatrix} cs & sn \\ -sn & cs \end{bmatrix}$$

where either

1. $cc = 0$ so that aa and dd are real eigenvalues of the matrix, or
2. $aa = dd$ and $bb*cc < 0$, so that $aa \pm$ sqrt($bb*cc$) are complex conjugate eigenvalues.

The routine was adjusted to reduce the risk of cancellation errors, when computing real eigenvalues, and to ensure, if possible, that abs(*rt1r*) ≥ abs(*rt2r*).

Input Parameters

a, b, c, d	`REAL` for `slanv2` `DOUBLE PRECISION` for `dlanv2`. On entry, elements of the input matrix.

Output Parameters

a, b, c, d	On exit, overwritten by the elements of the standardized Schur form.
rt1r, rt1i, rt2r, rt2i,	`REAL` for `slanv2` `DOUBLE PRECISION` for `dlanv2`. The real and imaginary parts of the eigenvalues. If the eigenvalues are a complex conjugate pair, *rt1i* > 0.

cs, sn REAL for slanv2
DOUBLE PRECISION for dlanv2.
Parameters of the rotation matrix.

?lapll

Measures the linear dependence of two vectors.

Syntax

```
call slapll ( n, x, incx, y, incy, ssmin )
call dlapll ( n, x, incx, y, incy, ssmin )
call clapll ( n, x, incx, y, incy, ssmin )
call zlapll ( n, x, incx, y, incy, ssmin )
```

Description

Given two column vectors x and y of length *n*, let

A = (x y) be the *n*-by-2 matrix.

The routine ?lapll first computes the QR factorization of A as $A = QR$ and then computes the SVD of the 2-by-2 upper triangular matrix R. The smaller singular value of R is returned in *ssmin*, which is used as the measurement of the linear dependency of the vectors x and y.

Input Parameters

n INTEGER. The length of the vectors x and y.

x REAL for slapll
DOUBLE PRECISION for dlapll
COMPLEX for clapll
COMPLEX*16 for zlapll
Array, DIMENSION (1+(*n*-1)*incx*).
On entry, *x* contains the *n*-vector x.

y REAL for slapll
DOUBLE PRECISION for dlapll
COMPLEX for clapll
COMPLEX*16 for zlapll
Array, DIMENSION (1+(*n*-1)*incy*). On entry, *y* contains the *n*-vector y.

incx	`INTEGER`. The increment between successive elements of *x*; *incx* > 0.
incy	`INTEGER`. The increment between successive elements of *y*; *incy* > 0.

Output Parameters

x	On exit, *x* is overwritten.
y	On exit, *y* is overwritten.
ssmin	`REAL` for `slapll/clapll` `DOUBLE PRECISION` for `dlapll/zlapll` The smallest singular value of the *n*-by-2 matrix $A = (x\ y)$.

?lapmt

Performs a forward or backward permutation of the columns of a matrix.

Syntax

```
call slapmt ( forwrd, m, n, x, ldx, k )
call dlapmt ( forwrd, m, n, x, ldx, k )
call clapmt ( forwrd, m, n, x, ldx, k )
call zlapmt ( forwrd, m, n, x, ldx, k )
```

Description

The routine `?lapmt` rearranges the columns of the *m*-by-*n* matrix *X* as specified by the permutation $k(1),k(2),...,k(n)$ of the integers $1,...,n$.

If *forwrd* = `.TRUE.`, forward permutation:

$X(*,k(j))$ is moved to $X(*,j)$ for $j= 1,2,...,n$.

If *forwrd* = `.FALSE.`, backward permutation:

$X(*,j)$ is moved to $X(*,k(j))$ for $j = 1,2,...,n$.

Input Parameters

forwrd	`LOGICAL`. If *forwrd* = `.TRUE.`, forward permutation If *forwrd* = `.FALSE.`, backward permutation
m	`INTEGER`. The number of rows of the matrix *X*. $m \geq 0$.
n	`INTEGER`. The number of columns of the matrix *X*. $n \geq 0$.
x	`REAL` for `slapmt` `DOUBLE PRECISION` for `dlapmt` `COMPLEX` for `clapmt` `COMPLEX*16` for `zlapmt` Array, `DIMENSION` (*ldx*,*n*). On entry, the *m*-by-*n* matrix *X*.
ldx	`INTEGER`. The leading dimension of the array *x*, $ldx \geq \max(1,m)$.
k	`INTEGER`. Array, `DIMENSION` (*n*). On entry, *k* contains the permutation vector.

Output Parameters

x	On exit, *x* contains the permuted matrix *X*.

?lapy2

Returns sqrt(x^2+y^2).

Syntax

```
val = slapy2 ( x, y )
val = dlapy2 ( x, y )
```

Description

The function `?lapy2` returns sqrt(x^2+y^2), avoiding unnecessary overflow or harmful underflow.

Input Parameters

x, y	`REAL` for `slapy2` `DOUBLE PRECISION` for `dlapy2` Specify the input values *x* and *y*.

Output Parameters

val	`REAL` for `slapy2` `DOUBLE PRECISION` for `dlapy2`. Value returned by the function.

?lapy3

Returns sqrt(x^2+y^2+z^2).

Syntax

```
val = slapy3 ( x, y, z )
val = dlapy3 ( x, y, z )
```

Description

The function `?lapy3` returns sqrt(x^2+y^2+z^2), avoiding unnecessary overflow or harmful underflow.

Input Parameters

x, y, z	`REAL` for `slapy3` `DOUBLE PRECISION` for `dlapy3` Specify the input values *x*, *y* and *z*.

Output Parameters

val	`REAL` for `slapy3` `DOUBLE PRECISION` for `dlapy3`. Value returned by the function.

?laqgb

Scales a general band matrix, using row and column scaling factors computed by ?gbequ.

Syntax

```
call slaqgb ( m, n, kl, ku, ab, ldab, r, c, rowcnd,
              colcnd, amax, equed )
call dlaqgb ( m, n, kl, ku, ab, ldab, r, c, rowcnd,
              colcnd, amax, equed )
call claqgb ( m, n, kl, ku, ab, ldab, r, c, rowcnd,
              colcnd, amax, equed )
call zlaqgb ( m, n, kl, ku, ab, ldab, r, c, rowcnd,
              colcnd, amax, equed )
```

Description

The routine equilibrates a general *m*-by-*n* band matrix *A* with *kl* subdiagonals and *ku* superdiagonals using the row and column scaling factors in the vectors *r* and *c*.

Input Parameters

m — INTEGER. The number of rows of the matrix *A*.
$m \ge 0$.

n — INTEGER. The number of columns of the matrix *A*.
$n \ge 0$.

kl — INTEGER. The number of subdiagonals within the band of *A*. $kl \ge 0$.

ku — INTEGER. The number of superdiagonals within the band of A. $ku \ge 0$.

ab — REAL for slaqgb
DOUBLE PRECISION for dlaqgb
COMPLEX for claqgb
COMPLEX*16 for zlaqgb
Array, DIMENSION (*ldab*,*n*). On entry, the matrix *A* in band storage, in rows 1 to $kl+ku+1$. The *j*-th column of A is stored in the *j*-th column of the array *ab* as follows: $ab(ku+1+i-j,j) = A(i,j)$ for
$\max(1,j-ku) \le i \le \min(m,j+kl)$.

ldab — INTEGER. The leading dimension of the array *ab*.
$lda \geq kl+ku+1$.

amax — REAL for slaqgb/claqgb
DOUBLE PRECISION for dlaqgb/zlaqgb
Absolute value of largest matrix entry.

Output Parameters

ab — On exit, the equilibrated matrix, in the same storage format as *A*.
See *equed* for the form of the equilibrated matrix.

r, c — REAL for slaqgb/claqgb
DOUBLE PRECISION for dlaqgb/zlaqgb
Arrays *r* (*m*), *c* (*n*). Contain the row and column scale factors for *A*, respectively.

rowcnd — REAL for slaqgb/claqgb
DOUBLE PRECISION for dlaqgb/zlaqgb
Ratio of the smallest *r*(*i*) to the largest *r*(*i*).

colcnd — REAL for slaqgb/claqgb
DOUBLE PRECISION for dlaqgb/zlaqgb
Ratio of the smallest *c*(*i*) to the largest *c*(*i*).

equed — CHARACTER*1.
Specifies the form of equilibration that was done.
If *equed* = 'N': No equilibration
If *equed* = 'R': Row equilibration, that is, *A* has been premultiplied by diag(*r*).
If *equed* = 'C': Column equilibration, that is, *A* has been postmultiplied by diag(*c*).
If *equed* = 'B': Both row and column equilibration, that is, *A* has been replaced by diag(*r*)*A*diag(*c*).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if row or column scaling should be done based on the ratio of the row or column scaling factors. If *rowcnd* < *thresh*, row scaling is done, and if *colcnd* < *thresh*, column scaling is done. *large* and *small* are threshold values used to decide if row scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, row scaling is done.

?laqge

Scales a general rectangular matrix, using row and column scaling factors computed by `?geequ`.

Syntax

```
call slaqge ( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
call dlaqge ( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
call claqge ( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
call zlaqge ( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
```

Description

The routine equilibrates a general *m*-by-*n* matrix *A* using the row and scaling factors in the vectors *r* and *c*.

Input Parameters

m — `INTEGER`. The number of rows of the matrix *A*.
$m \geq 0$.

n — `INTEGER`. The number of columns of the matrix *A*.
$n \geq 0$.

a — `REAL` for `slaqge`
`DOUBLE PRECISION` for `dlaqge`
`COMPLEX` for `claqge`
`COMPLEX*16` for `zlaqge`
Array, `DIMENSION` (*lda*,*n*). On entry, the *m*-by-*n* matrix *A*.

lda — `INTEGER`. The leading dimension of the array *A*.
$lda \geq \max(m,1)$.

r — `REAL` for `slanqge/claqge`
`DOUBLE PRECISION` for `dlaqge/zlaqge`
Array, `DIMENSION` (*m*). The row scale factors for *A*.

c — `REAL` for `slanqge/claqge`
`DOUBLE PRECISION` for `dlaqge/zlaqge`
Array, `DIMENSION` (*n*). The column scale factors for *A*.

rowcnd REAL for slanqge/claqge
DOUBLE PRECISION for dlaqge/zlaqge
Ratio of the smallest *r*(*i*) to the largest *r*(*i*).

colcnd REAL for slanqge/claqge
DOUBLE PRECISION for dlaqge/zlaqge
Ratio of the smallest *c*(*i*) to the largest *c*(*i*).

amax REAL for slanqge/claqge
DOUBLE PRECISION for dlaqge/zlaqge
Absolute value of largest matrix entry.

Output Parameters

a On exit, the equilibrated matrix.
See *equed* for the form of the equilibrated matrix.

equed CHARACTER*1.
Specifies the form of equilibration that was done.
If *equed* = 'N': No equilibration
If *equed* = 'R': Row equilibration, that is, *A* has been premultiplied by diag(*r*).
If *equed* = 'C': Column equilibration, that is, *A* has been postmultiplied by diag(*c*).
If *equed* = 'B': Both row and column equilibration, that is, *A* has been replaced by diag(*r*)*A*diag(*c*).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if row or column scaling should be done based on the ratio of the row or column scaling factors. If *rowcnd* < *thresh*, row scaling is done, and if *colcnd* < *thresh*, column scaling is done. *large* and *small* are threshold values used to decide if row scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, row scaling is done.

?laqp2

Computes a QR factorization with column pivoting of the matrix block.

Syntax

```
call slaqp2 ( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
call dlaqp2 ( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
call claqp2 ( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
call zlaqp2 ( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
```

Description

The routine computes a *QR* factorization with column pivoting of the block A(*offset*+1:*m*,1:*n*). The block A(1:*offset*,1:*n*) is accordingly pivoted, but not factorized.

Input Parameters

m — INTEGER. The number of rows of the matrix A.
m ≥ 0.

n — INTEGER. The number of columns of the matrix A.
n ≥ 0.

offset — INTEGER. The number of rows of the matrix A that must be pivoted but no factorized. *offset* ≥ 0.

a — REAL for slaqp2
DOUBLE PRECISION for dlaqp2
COMPLEX for claqp2
COMPLEX*16 for zlaqp2
Array, DIMENSION (*lda*,*n*). On entry, the *m*-by-*n* matrix A.

lda — INTEGER. The leading dimension of the array A. *lda* ≥ max(1,*m*).

jpvt — INTEGER .
Array, DIMENSION (*n*). On entry, if *jpvt*(*i*) ≠ 0, the *i*-th column of A is permuted to the front of $A*P$ (a leading column); if *jpvt*(*i*) = 0, the *i*-th column of A is a free column.

vn1, vn2	REAL for slaqp2/claqp2 DOUBLE PRECISION for dlaqp2/zlaqp2 Arrays, DIMENSION (*n*) each. Contain the vectors with the partial and exact column norms, respectively.
work	REAL for slaqp2 DOUBLE PRECISION for dlaqp2 COMPLEX for claqp2 COMPLEX*16 for zlaqp2 Workspace array, DIMENSION (*n*).

Output Parameters

a	On exit, the upper triangle of block *A*(*offset*+1:*m*,1:*n*) is the triangular factor obtained; the elements in block *A*(*offset*+1:*m*,1:*n*) below the diagonal, together with the array *tau*, represent the orthogonal matrix *Q* as a product of elementary reflectors. Block *A*(1:*offset*,1:*n*) has been accordingly pivoted, but not factorized.
jpvt	On exit, if *jpvt*(*i*) = *k*, then the *i*-th column of *A***P* was the *k*-th column of *A*.
tau	REAL for slaqp2 DOUBLE PRECISION for dlaqp2 COMPLEX for claqp2 COMPLEX*16 for zlaqp2 Array, DIMENSION (min(*m*,*n*)). The scalar factors of the elementary reflectors.
vn1, vn2	Contain the vectors with the partial and exact column norms, respectively.

?laqps

Computes a step of QR factorization with column pivoting of a real m-by-n *matrix A by using BLAS level 3.*

Syntax

```
call slaqps ( m, n, offset, nb, kb, a, lda, jpvt, tau,
              vn1, vn2, auxv, f, ldf )
call dlaqps ( m, n, offset, nb, kb, a, lda, jpvt, tau,
              vn1, vn2, auxv, f, ldf )
```

```
call claqps ( m, n, offset, nb, kb, a, lda, jpvt, tau,
              vn1, vn2, auxv, f, ldf )
call zlaqps ( m, n, offset, nb, kb, a, lda, jpvt, tau,
              vn1, vn2, auxv, f, ldf )
```

Description

This routine computes a step of *QR* factorization with column pivoting of a real *m*-by-*n* matrix *A* by using BLAS level 3. The routine tries to factorize *nb* columns from *A* starting from the row *offset*+1, and updates all of the matrix with BLAS level 3 routine ?gemm.

In some cases, due to catastrophic cancellations, ?laqps cannot factorize *nb* columns. Hence, the actual number of factorized columns is returned in *kb*.

Block *A*(1:*offset*,1:*n*) is accordingly pivoted, but not factorized.

Input Parameters

m	INTEGER. The number of rows of the matrix *A*. $m \geq 0$.
n	INTEGER. The number of columns of the matrix *A*. $n \geq 0$.
offset	INTEGER. The number of rows of *A* that have been factorized in previous steps.
nb	INTEGER. The number of columns to factorize.
a	REAL for slaqps DOUBLE PRECISION for dlaqps COMPLEX for claqps COMPLEX*16 for zlaqps Array, DIMENSION (*lda*,*n*). On entry, the *m*-by-*n* matrix *A*.
lda	INTEGER. The leading dimension of the array *a*. $lda \geq \max(1,m)$.
jpvt	INTEGER. Array, DIMENSION (*n*). If *jpvt*(*i*) = *k* then column *k* of the full matrix *A* has been permuted into position *i* in *AP*.

vn1, vn2	`REAL` for `slaqps/claqps` `DOUBLE PRECISION` for `dlaqps/zlaqps` Arrays, `DIMENSION` (*n*) each. Contain the vectors with the partial and exact column norms, respectively.
auxv	`REAL` for `slaqps` `DOUBLE PRECISION` for `dlaqps` `COMPLEX` for `claqps` `COMPLEX*16` for `zlaqps` Array, `DIMENSION` *(nb)*. Auxiliary vector.
f	`REAL` for `slaqps` `DOUBLE PRECISION` for `dlaqps` `COMPLEX` for `claqps` `COMPLEX*16` for `zlaqps` Array, `DIMENSION` (*ldf*,*nb*). Matrix $F' = L*Y'*A$.
ldf	`INTEGER`. The leading dimension of the array *f*. *ldf* ≥ max(1,*n*).

Output Parameters

kb	`INTEGER`. The number of columns actually factorized.
a	On exit, block *A*(*offset*+1:*m*,1:*kb*) is the triangular factor obtained and block *A*(1:*offset*,1:*n*) has been accordingly pivoted, but no factorized. The rest of the matrix, block *A*(*offset*+1:*m*,*kb*+1:*n*) has been updated.
jpvt	`INTEGER` array, `DIMENSION` (*n*). If *jpvt*(*i*) = *k* then column *k* of the full matrix *A* has been permuted into position *i* in *AP*.
tau	`REAL` for `slaqps` `DOUBLE PRECISION` for `dlaqps` `COMPLEX` for `claqps` `COMPLEX*16` for `zlaqps` Array, `DIMENSION` (*kb*). The scalar factors of the elementary reflectors.
vn1, vn2	The vectors with the partial and exact column norms, respectively.
auxv	Auxiliary vector.
f	Matrix $F' = L*Y'*A$.

?laqsb

Scales a symmetric/Hermitian band matrix, using scaling factors computed by ?pbequ.

Syntax

```
call slaqsb ( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call dlaqsb ( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call claqsb ( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call zlaqsb ( uplo, n, kd, ab, ldab, s, scond, amax, equed )
```

Description

The routine equilibrates a symmetric band matrix *A* using the scaling factors in the vector *s*.

Input Parameters

uplo — CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored.
If *uplo* = 'U': upper triangular.
If *uplo* = 'L': lower triangular.

n — INTEGER. The order of the matrix *A*.
$n \geq 0$.

kd — INTEGER. The number of super-diagonals of the matrix *A* if *uplo* = 'U', or the number of sub-diagonals if *uplo* = 'L'. $kd \geq 0$.

ab — REAL for slaqsb
DOUBLE PRECISION for dlaqsb
COMPLEX for claqsb
COMPLEX*16 for zlaqsb
Array, DIMENSION (*ldab*,*n*). On entry, the upper or lower triangle of the symmetric band matrix *A*, stored in the first *kd*+1 rows of the array. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:
if *uplo* = 'U', $ab(kd+1+i-j,j) = A(i,j)$ for
$\max(1,j-kd) \leq i \leq j$;
if *uplo* = 'L', $ab(1+i-j,j) = A(i,j)$ for
$j \leq i \leq \min(n,j+kd)$.

ldab — INTEGER. The leading dimension of the array *ab*.
ldab ≥ *kd*+1.

scond — REAL for slaqsb/claqsb
DOUBLE PRECISION for dlaqsb/zlaqsb
Ratio of the smallest *s*(*i*) to the largest *s*(*i*).

amax — REAL for slaqsb/claqsb
DOUBLE PRECISION for dlaqsb/zlaqsb
Absolute value of largest matrix entry.

Output Parameters

ab — On exit, if *info* = 0, the triangular factor *U* or *L* from the Cholesky factorization $A = U' U$ or $A = L L'$ of the band matrix *A*, in the same storage format as *A*.

s — REAL for slaqsb/claqsb
DOUBLE PRECISION for dlaqsb/zlaqsb
Array, DIMENSION (*n*). The scale factors for *A*.

equed — CHARACTER*1.
Specifies whether or not equilibration was done.
If *equed* = 'N': No equilibration.
If *equed* = 'Y': Equilibration was done, that is, *A* has been replaced by diag(*s*)**A**diag(*s*).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If *scond* < *thresh*, scaling is done. *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, scaling is done.

?laqsp

Scales a symmetric/Hermitian matrix in packed storage, using scaling factors computed by `?ppequ`.

Syntax

```
call slaqsp ( uplo, n, ap, s, scond, amax, equed )
call dlaqsp ( uplo, n, ap, s, scond, amax, equed )
call claqsp ( uplo, n, ap, s, scond, amax, equed )
call zlaqsp ( uplo, n, ap, s, scond, amax, equed )
```

Description

The routine `?laqsp` equilibrates a symmetric matrix A using the scaling factors in the vector *s*.

Internal Parameters

uplo — `CHARACTER*1`. Specifies whether the upper or lower triangular part of the symmetric matrix A is stored.
If *uplo* = 'U': upper triangular.
If *uplo* = 'L': lower triangular.

n — `INTEGER`. The order of the matrix A.
$n \geq 0$.

ap — `REAL` for `slaqsp`
`DOUBLE PRECISION` for `dlaqsp`
`COMPLEX` for `claqsp`
`COMPLEX*16` for `zlaqsp`
Array, `DIMENSION` $(n(n+1)/2)$. On entry, the upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array *ap* as follows:
if *uplo* = 'U', $ap(i + (j-1)j/2) = A(i,j)$ for $1 \leq i \leq j$;
if *uplo* = 'L', $ap(i + (j-1)(2n-j)/2) = A(i,j)$ for $j \leq i \leq n$.

s — `REAL` for `slaqsp/claqsp`
`DOUBLE PRECISION` for `dlaqsp/zlaqsp`
Array, `DIMENSION` (n). The scale factors for A.

scond REAL for slaqsp/claqsp
DOUBLE PRECISION for dlaqsp/zlaqsp
Ratio of the smallest s(*i*) to the largest s(*i*).

amax REAL for slaqsp/claqsp
DOUBLE PRECISION for dlaqsp/zlaqsp
Absolute value of largest matrix entry.

Output Parameters

ap On exit, the equilibrated matrix: diag(s)**A**diag(s), in the same storage format as *A*.

equed CHARACTER*1.
Specifies whether or not equilibration was done.
If *equed* = 'N': No equilibration.
If *equed* = 'Y': Equilibration was done, that is, *A* has been replaced by diag(s)**A**diag(s).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If *scond* < *thresh*, scaling is done. *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, scaling is done.

?laqsy

Scales a symmetric/Hermitian matrix, using scaling factors computed by ?poequ.

Syntax

```
call slaqsy ( uplo, n, a, lda, s, scond, amax, equed )
call dlaqsy ( uplo, n, a, lda, s, scond, amax, equed )
call claqsy ( uplo, n, a, lda, s, scond, amax, equed )
call zlaqsy ( uplo, n, a, lda, s, scond, amax, equed )
```

Description

The routine equilibrates a symmetric matrix *A* using the scaling factors in the vector *s*.

Input Parameters

uplo — CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored.
If *uplo* = 'U': upper triangular.
If *uplo* = 'L': lower triangular.

n — INTEGER. The order of the matrix *A*.
$n \geq 0$.

a — REAL for slaqsy
DOUBLE PRECISION for dlaqsy
COMPLEX for claqsy
COMPLEX*16 for zlaqsy
Array, DIMENSION (*lda*,*n*). On entry, the symmetric matrix *A*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda — INTEGER. The leading dimension of the array *a*.
$lda \geq \max(n,1)$.

s — REAL for slaqsy/claqsy
DOUBLE PRECISION for dlaqsy/zlaqsy
Array, DIMENSION (*n*). The scale factors for *A*.

scond — REAL for slaqsy/claqsy
DOUBLE PRECISION for dlaqsy/zlaqsy
Ratio of the smallest *s*(*i*) to the largest *s*(*i*).

amax — REAL for slaqsy/claqsy
DOUBLE PRECISION for dlaqsy/zlaqsy
Absolute value of largest matrix entry.

Output Parameters

a — On exit, if *equed* = 'Y', the equilibrated matrix: diag(*s*)**A**diag(*s*).

equed CHARACTER*1.
Specifies whether or not equilibration was done.
If *equed* = 'N': No equilibration.
If *equed* = 'Y': Equilibration was done, i.e., *A* has been replaced by diag(*s*)**A**diag(*s*).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If *scond* < *thresh*, scaling is done. *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, scaling is done.

?laqtr

Solves a real quasi-triangular system of equations, or a complex quasi-triangular system of special form, in real arithmetic.

Syntax

```
call slaqtr ( ltran, lreal, n, t, ldt, b, w, scale, x,
              work, info )
call dlaqtr ( ltran, lreal, n, t, ldt, b, w, scale, x,
              work, info )
```

Description

The routine ?laqtr solves the real quasi-triangular system

op(T) * p = *scale** c, if *lreal* = .TRUE.

or the complex quasi-triangular systems

op(T + $\mathbf{i}B$)*(p+$\mathbf{i}q$) = *scale**(c+$\mathbf{i}d$), if *lreal* = .FALSE.

in real arithmetic, where T is upper quasi-triangular.

If *lreal* = .FALSE., then the first diagonal block of T must be 1-by-1, B is the specially structured matrix

$$B = \begin{bmatrix} b_1 & b_2 & \dots & \dots & b_n \\ & w & & & \\ & & w & & \\ & & & \dots & \\ & & & & w \end{bmatrix}$$

op$(A) = A$ or A', A' denotes the conjugate transpose of matrix A.

On input,

$$x = \begin{bmatrix} c \\ d \end{bmatrix}, \text{ on output } x = \begin{bmatrix} p \\ q \end{bmatrix}$$

This routine is designed for the condition number estimation in routine `?trsna`.

Input Parameters

ltran — `LOGICAL`.
On entry, *ltran* specifies the option of conjugate transpose:
= `.FALSE.`, $\mathrm{op}(T + iB) = T + iB$,
= `.TRUE.`, $\mathrm{op}(T + iB) = (T + iB)'$.

lreal — `LOGICAL`.
On entry, *lreal* specifies the input matrix structure:
= `.FALSE.`, the input is complex
= `.TRUE.`, the input is real.

n — `INTEGER`. On entry, *n* specifies the order of $T + iB$.
$n \ge 0$.

t — `REAL` for `slaqtr`
`DOUBLE PRECISION` for `dlaqtr`
Array, dimension (*ldt*,*n*). On entry, *t* contains a matrix in Schur canonical form. If *lreal* = `.FALSE.`, then the first diagonal block of *t* must be 1-by-1.

ldt — `INTEGER`. The leading dimension of the matrix *T*.
$ldt \ge \max(1,n)$.

b REAL for slaqtr
DOUBLE PRECISION for dlaqtr
Array, dimension *(n)*. On entry, *b* contains the elements to form the matrix *B* as described above. If *lreal* = .TRUE., *b* is not referenced.

w REAL for slaqtr
DOUBLE PRECISION for dlaqtr
On entry, *w* is the diagonal element of the matrix *B*. If *lreal* = .TRUE., *w* is not referenced.

x REAL for slaqtr
DOUBLE PRECISION for dlaqtr
Array, dimension (2*n*). On entry, *x* contains the right hand side of the system.

work REAL for slaqtr
DOUBLE PRECISION for dlaqtr
Workspace array, dimension *(n)*.

Output Parameters

scale REAL for slaqtr
DOUBLE PRECISION for dlaqtr
On exit, *scale* is the scale factor.

x On exit, *x* is overwritten by the solution.

info INTEGER.
If *info* = 0: successful exit.
If *info* = 1: the some diagonal 1-by-1 block has been perturbed by a small number smin to keep nonsingularity.
If *info* = 2: the some diagonal 2-by-2 block has been perturbed by a small number in ?laln2 to keep nonsingularity.

NOTE. *In the interests of speed, this routine does not check the inputs for errors.*

?lar1v

Computes the (scaled) `r`*-th column of the inverse of the submatrix in rows* `b1` *through* `bn` *of the tridiagonal matrix* $LDL^T - \sigma I$.

Syntax

```
call slar1v ( n, b1, bn, sigma, d, l, ld, lld, gersch, z,
              ztz, mingma, r, isuppz, work )
call dlar1v ( n, b1, bn, sigma, d, l, ld, lld, gersch, z,
              ztz, mingma, r, isuppz, work )
call clar1v ( n, b1, bn, sigma, d, l, ld, lld, gersch, z,
              ztz, mingma, r, isuppz, work )
call zlar1v ( n, b1, bn, sigma, d, l, ld, lld, gersch, z,
              ztz, mingma, r, isuppz, work )
```

Description

The routine `?lar1v` computes the (scaled) `r`-th column of the inverse of the submatrix in rows *b1* through *bn* of the tridiagonal matrix
$LDL^T - \sigma * I$.
The following steps accomplish this computation :

1. Stationary *qd* transform, $LDL^T - \sigma * I = L(+)\ D(+)\ L(+)^T$
2. Progressive *qd* transform, $LDL^T - \sigma * I = U(-)\ D(-)\ U(-)^T$,
3. Computation of the diagonal elements of the inverse of $LDL^T - \sigma * I$ by combining the above transforms, and choosing `r` as the index where the diagonal of the inverse is (one of the) largest in magnitude.
4. Computation of the (scaled) `r`-th column of the inverse using the twisted factorization obtained by combining the top part of the stationary and the bottom part of the progressive transform.

Input Parameters

n — `INTEGER`. The order of the matrix LDL^T.

b1 — `INTEGER`. First index of the submatrix of LDL^T.

bn — `INTEGER`. Last index of the submatrix of LDL^T.

sigma REAL for slar1v/clar1v
DOUBLE PRECISION for dlar1v/zlar1v
The shift. Initially, when *r* = 0, *sigma* should be a good approximation to an eigenvalue of LDL^T.

l REAL for slar1v/clar1v
DOUBLE PRECISION for dlar1v/zlar1v
Array, DIMENSION (*n*-1). The (*n*-1) subdiagonal elements of the unit bidiagonal matrix *L*, in elements 1 to *n*-1.

d REAL for slar1v/clar1v
DOUBLE PRECISION for dlar1v/zlar1v
Array, DIMENSION *(n)*. The *n* diagonal elements of the diagonal matrix *D*.

ld REAL for slar1v/clar1v
DOUBLE PRECISION for dlar1v/zlar1v
Array, DIMENSION (*n*-1). The *n*-1 elements L_i*D_i.

lld REAL for slar1v/clar1v
DOUBLE PRECISION for dlar1v/zlar1v
Array, DIMENSION (*n*-1). The *n*-1 elements $L_i*L_i*D_i$.

gersch REAL for slar1v/clar1v
DOUBLE PRECISION for dlar1v/zlar1v
Array, DIMENSION (2*n*). The *n* Gerschgorin intervals. These are used to restrict the initial search for *r*, when *r* is input as 0.

r INTEGER.
Initially *r* should be input to be 0 and is then output as the index where the diagonal element of the inverse is largest in magnitude. In later iterations, this same value of *r* should be input.

work REAL for slar1v/clar1v
DOUBLE PRECISION for dlar1v/zlar1v
Workspace array, DIMENSION (4*n*).

Output Parameters

z REAL for slar1v
DOUBLE PRECISION for dlar1v
COMPLEX for clar1v
COMPLEX*16 for zlar1v
Array, DIMENSION *(n)*. The (scaled) *r*-th column of the inverse. *z(r)* is returned to be 1.

ztz — REAL for slar1v/clar1v
DOUBLE PRECISION for dlar1v/zlar1v
The square of the norm of *z*.

mingma — REAL for slar1v/clar1v
DOUBLE PRECISION for dlar1v/zlar1v
The reciprocal of the largest (in magnitude) diagonal element of the inverse of $LDL^T - \sigma * I$.

r — On output, *r* is the index where the diagonal element of the inverse is largest in magnitude.

isuppz — INTEGER.
Array, DIMENSION (2). The support of the vector in *z*, that is, the vector *z* is nonzero only in elements *isuppz*(1) through *isuppz*(2).

?lar2v

Applies a vector of plane rotations with real cosines and real/complex sines from both sides to a sequence of 2-by-2 symmetric/Hermitian matrices.

Syntax

```
call slar2v ( n, x, y, z, incx, c, s, incc )
call dlar2v ( n, x, y, z, incx, c, s, incc )
call clar2v ( n, x, y, z, incx, c, s, incc )
call zlar2v ( n, x, y, z, incx, c, s, incc )
```

Description

The routine ?lar2v applies a vector of real/complex plane rotations with real cosines from both sides to a sequence of 2-by-2 real symmetric or complex Hermitian matrices, defined by the elements of the vectors x, y and z. For $i = 1,2,\ldots,n$

$$\begin{bmatrix} x_i & z_i \\ \mathrm{conjg}(z_i) & y_i \end{bmatrix} := \begin{bmatrix} c(i) & \mathrm{conjg}(s(i)) \\ -s(i) & c(i) \end{bmatrix} \begin{bmatrix} x_i & z_i \\ \mathrm{conjg}(z_i) & y_i \end{bmatrix} \begin{bmatrix} c(i) & -\mathrm{conjg}(s(i)) \\ s(i) & c(i) \end{bmatrix}$$

Input Parameters

n	`INTEGER`. The number of plane rotations to be applied.
x, y, z	`REAL` for `slar2v` `DOUBLE PRECISION` for `dlar2v` `COMPLEX` for `clar2v` `COMPLEX*16` for `zlar2v` Arrays, `DIMENSION` (1+(*n*-1)*`incx`) each. Contain the vectors *x*, *y* and *z*, respectively. For all flavors of `?lar2v`, elements of *x* and *y* are assumed to be real.
incx	`INTEGER`. The increment between elements of *x*, *y*, and *z*. *incx* > 0.
c	`REAL` for `slar2v`/`clar2v` `DOUBLE PRECISION` for `dlar2v`/`zlar2v` Array, `DIMENSION` (1+(*n*-1)*`incc`). The cosines of the plane rotations.
s	`REAL` for `slar2v` `DOUBLE PRECISION` for `dlar2v` `COMPLEX` for `clar2v` `COMPLEX*16` for `zlar2v` Array, `DIMENSION` (1+(*n*-1)*`incc`). The sines of the plane rotations.
incc	`INTEGER`. The increment between elements of *c* and *s*. *incc* > 0.

Output Parameters

x, y, z	Vectors *x*, *y* and *z*, containing the results of transform.

?larf

Applies an elementary reflector to a general rectangular matrix.

Syntax

```
call slarf ( side, m, n, v, incv, tau, c, ldc, work )
call dlarf ( side, m, n, v, incv, tau, c, ldc, work )
call clarf ( side, m, n, v, incv, tau, c, ldc, work )
call zlarf ( side, m, n, v, incv, tau, c, ldc, work )
```

Description

The routine applies a real/complex elementary reflector *H* to a real/complex *m*-by-*n* matrix *C*, from either the left or the right. *H* is represented in the form
H = *I* - *tau* * *v* * *v*',
where *tau* is a real/complex scalar and *v* is a real/complex vector.

If *tau* = 0, then *H* is taken to be the unit matrix.
For clarf/zlarf, to apply *H'* (the conjugate transpose of *H*), supply conjg(*tau*) instead of *tau*.

Input Parameters

side	CHARACTER*1. If *side* = 'L': form *H* * *C* If *side* = 'R': form *C* * *H*.
m	INTEGER. The number of rows of the matrix *C*.
n	INTEGER. The number of columns of the matrix *C*.
v	REAL for slarf DOUBLE PRECISION for dlarf COMPLEX for clarf COMPLEX*16 for zlarf Array, DIMENSION (1 + (*m*-1)*abs(*incv*)) if *side* = 'L' or (1 + (*n*-1)*abs(*incv*)) if *side* = 'R'. The vector *v* in the representation of *H*. *v* is not used if *tau* = 0.
incv	INTEGER. The increment between elements of *v*. *incv* ≠ 0.
tau	REAL for slarf DOUBLE PRECISION for dlarf COMPLEX for clarf COMPLEX*16 for zlarf The value *tau* in the representation of *H*.
c	REAL for slarf DOUBLE PRECISION for dlarf COMPLEX for clarf COMPLEX*16 for zlarf Array, DIMENSION (*ldc*,*n*). On entry, the *m*-by-*n* matrix *C*.

ldc	INTEGER. The leading dimension of the array *c*. *ldc* ≥ max(1,*m*).
work	REAL for slarf DOUBLE PRECISION for dlarf COMPLEX for clarf COMPLEX*16 for zlarf Workspace array, DIMENSION (*n*) if *side* = 'L' or (*m*) if *side* = 'R'.

Output Parameters

c	On exit, *c* is overwritten by the matrix *H***C* if *side* = 'L', or *C***H* if *side* = 'R'.

?larfb

Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.

Syntax

```
call slarfb ( side, trans, direct, storev, m, n, k, v,
              ldv, t, ldt, c, ldc, work, ldwork )
call dlarfb ( side, trans, direct, storev, m, n, k, v,
              ldv, t, ldt, c, ldc, work, ldwork )
call clarfb ( side, trans, direct, storev, m, n, k, v,
              ldv, t, ldt, c, ldc, work, ldwork )
call zlarfb ( side, trans, direct, storev, m, n, k, v,
              ldv, t, ldt, c, ldc, work, ldwork )
```

Description

The routine ?larfb applies a complex block reflector *H* or its transpose *H*′ to a complex *m*-by-*n* matrix *C* from either left or right.

Input Parameters

side	CHARACTER*1. If *side* = 'L': apply H or H' from the left If *side* = 'R': apply H or H' from the right
trans	CHARACTER*1. If *trans* = 'N': apply H (No transpose) If *trans* = 'C': apply H' (Conjugate transpose)
direct	CHARACTER*1. Indicates how H is formed from a product of elementary reflectors If *direct* = 'F': $H = H(1)\ H(2) \ldots H(k)$ (forward) If *direct* = 'B': $H = H(k) \ldots H(2)\ H(1)$ (backward)
storev	CHARACTER*1. Indicates how the vectors which define the elementary reflectors are stored: If *storev* = 'C': Column-wise If *storev* = 'R': Row-wise
m	INTEGER. The number of rows of the matrix C.
n	INTEGER. The number of columns of the matrix C.
k	INTEGER. The order of the matrix T (equal to the number of elementary reflectors whose product defines the block reflector).
v	REAL for slarfb DOUBLE PRECISION for dlarfb COMPLEX for clarfb COMPLEX*16 for zlarfb Array, DIMENSION (*ldv*, *k*) if *storev* = 'C' (*ldv*, *m*) if *storev* = 'R' and *side* = 'L' (*ldv*, *n*) if *storev* = 'R' and *side* = 'R' The matrix V.
ldv	INTEGER. The leading dimension of the array v. If *storev* = 'C' and *side* = 'L', *ldv* $\geq$ max(1,*m*); if *storev* = 'C' and *side* = 'R', *ldv* $\geq$ max(1,*n*); if *storev* = 'R', *ldv* $\geq$ *k*.
t	REAL for slarfb DOUBLE PRECISION for dlarfb COMPLEX for clarfb

	`COMPLEX*16` for `zlarfb` Array, `DIMENSION` (*ldt*,*k*). Contains the triangular *k*-by-*k* matrix *T* in the representation of the block reflector.
ldt	`INTEGER`. The leading dimension of the array *t*. *ldt* ≥ *k*.
c	`REAL` for `slarfb` `DOUBLE PRECISION` for `dlarfb` `COMPLEX` for `clarfb` `COMPLEX*16` for `zlarfb` Array, `DIMENSION` (*ldc*,*n*). On entry, the *m*-by-*n* matrix *C*.
ldc	`INTEGER`. The leading dimension of the array *c*. *ldc* ≥ max(1,*m*).
work	`REAL` for `slarfb` `DOUBLE PRECISION` for `dlarfb` `COMPLEX` for `clarfb` `COMPLEX*16` for `zlarfb` Workspace array, `DIMENSION` (*ldwork*, *k*).
ldwork	`INTEGER`. The leading dimension of the array *work*. If *side* = 'L', *ldwork* ≥ max(1, *n*); if *side* = 'R', *ldwork* ≥ max(1, *m*).

Output parameters

c	On exit, *c* is overwritten by $H*C$ or $H'*C$ or $C*H$ or $C*H'$.

?larfg

Generates an elementary reflector (Householder matrix).

Syntax

```
call slarfg ( n, alpha, x, incx, tau )
call dlarfg ( n, alpha, x, incx, tau )
```

```
call clarfg ( n, alpha, x, incx, tau )
call zlarfg ( n, alpha, x, incx, tau )
```

Description

The routine `?larfg` generates a real/complex elementary reflector *H* of order *n*, such that

$$H'*\begin{bmatrix} alpha \\ x \end{bmatrix} = \begin{bmatrix} beta \\ 0 \end{bmatrix}, \quad H'*H = I,$$

where *alpha* and *beta* are scalars (with *beta* real for all flavors), and *x* is an (*n*-1)-element real/complex vector. *H* is represented in the form

$$H = I - tau*\begin{bmatrix} 1 \\ v \end{bmatrix}*\begin{bmatrix} 1 & v' \end{bmatrix}$$

where *tau* is a real/complex scalar and *v* is a real/complex (*n*-1)-element vector. Note that for `clarfg/zlarfg`, *H* is not Hermitian.

If the elements of *x* are all zero (and, for complex flavors, *alpha* is real), then *tau* = 0 and *H* is taken to be the unit matrix.

Otherwise, $1 \le tau \le 2$ (for real flavors), or
$1 \le \mathrm{Re}(tau) \le 2$ and $\mathrm{abs}(tau-1) \le 1$ (for complex flavors).

Input Parameters

n — `INTEGER`. The order of the elementary reflector.

alpha — `REAL` for `slarfg`
`DOUBLE PRECISION` for `dlarfg`
`COMPLEX` for `clarfg`
`COMPLEX*16` for `zlarfg`
On entry, the value *alpha*.

x — `REAL` for `slarfg`
`DOUBLE PRECISION` for `dlarfg`
`COMPLEX` for `clarfg`

	`COMPLEX*16` for `zlarfg` Array, `DIMENSION` (1+(*n*-2)*abs(*incx*)). On entry, the vector *x*.
incx	`INTEGER`. The increment between elements of *x*. *incx* > 0.

Output Parameters

alpha	On exit, it is overwritten with the value *beta*.
x	On exit, it is overwritten with the vector *v*.
tau	`REAL` for `slarfg` `DOUBLE PRECISION` for `dlarfg` `COMPLEX` for `clarfg` `COMPLEX*16` for `zlarfg` The value *tau*.

?larft

Forms the triangular factor T of a block reflector $H = I - VTV^H$.

Syntax

```
call slarft ( direct, storev, n, k, v, ldv, tau, t, ldt )
call dlarft ( direct, storev, n, k, v, ldv, tau, t, ldt )
call clarft ( direct, storev, n, k, v, ldv, tau, t, ldt )
call zlarft ( direct, storev, n, k, v, ldv, tau, t, ldt )
```

Description

The routine `?larft` forms the triangular factor *T* of a real/complex block reflector *H* of order *n*, which is defined as a product of *k* elementary reflectors.

If *direct* = 'F', $H = H(1)\ H(2) \ldots H(k)$ and *T* is upper triangular;

If *direct* = 'B', $H = H(k) \ldots H(2)\ H(1)$ and *T* is lower triangular.

If *storev* = 'C', the vector which defines the elementary reflector H(*i*) is stored in the *i*-th column of the array *v*, and $H = I - V^*T^*V'$.

If *storev* = 'R', the vector which defines the elementary reflector H(*i*) is stored in the *i*-th row of the array *v*, and $H = I - V' * T * V$.

Input Parameters

direct	CHARACTER*1. Specifies the order in which the elementary reflectors are multiplied to form the block reflector: = 'F': $H = H(1)\ H(2)\ .\ .\ .\ H(k)$ (forward) = 'B': $H = H(k)\ .\ .\ .\ H(2)\ H(1)$ (backward)
storev	CHARACTER*1. Specifies how the vectors which define the elementary reflectors are stored (see also *Application Notes* below): = 'C': column-wise = 'R': row-wise.
n	INTEGER. The order of the block reflector *H*. $n \geq 0$.
k	INTEGER. The order of the triangular factor *T* (equal to the number of elementary reflectors). $k \geq 1$.
v	REAL for slarft DOUBLE PRECISION for dlarft COMPLEX for clarft COMPLEX*16 for zlarft Array, DIMENSION (*ldv*, *k*) if *storev* = 'C' or (*ldv*, *n*) if *storev* = 'R'. The matrix *V*.
ldv	INTEGER. The leading dimension of the array *v*. If *storev* = 'C', $ldv \geq \max(1,n)$; if *storev* = 'R', $ldv \geq k$.
tau	REAL for slarft DOUBLE PRECISION for dlarft COMPLEX for clarft COMPLEX*16 for zlarft Array, DIMENSION (*k*). *tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*).
ldt	INTEGER. The leading dimension of the output array *t*. $ldt \geq k$.

Output Parameters

t — `REAL` for `slarft`
`DOUBLE PRECISION` for `dlarft`
`COMPLEX` for `clarft`
`COMPLEX*16` for `zlarft`
Array, `DIMENSION` (*ldt*,*k*). The *k*-by-*k* triangular factor *T* of the block reflector. If *direct* = 'F', *T* is upper triangular; if *direct* = 'B', *T* is lower triangular. The rest of the array is not used.

v — The matrix *V*.

Application Notes

The shape of the matrix *V* and the storage of the vectors which define the *H*(*i*) is best illustrated by the following example with *n* = 5 and *k* = 3. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

direct = 'F' and *storev* = 'C':

$$\begin{bmatrix} 1 & & \\ v_1 & 1 & \\ v_1 & v_2 & 1 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \end{bmatrix}$$

direct = 'F' and *storev* = 'R':

$$\begin{bmatrix} 1 & v_1 & v_1 & v_1 & v_1 \\ & 1 & v_2 & v_2 & v_2 \\ & & 1 & v_3 & v_3 \end{bmatrix}$$

direct = 'B' and *storev* = 'C':

$$\begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ 1 & v_2 & v_3 \\ & 1 & v_3 \\ & & 1 \end{bmatrix}$$

direct = 'B' and *storev* = 'R':

$$\begin{bmatrix} v_1 & v_1 & 1 & & \\ v_2 & v_2 & v_2 & 1 & \\ v_3 & v_3 & v_3 & v_3 & 1 \end{bmatrix}$$

?larfx

Applies an elementary reflector to a general rectangular matrix, with loop unrolling when the reflector has order ≤ 10.

Syntax

```
call slarfx ( side, m, n, v, tau, c, ldc, work )
call dlarfx ( side, m, n, v, tau, c, ldc, work )
call clarfx ( side, m, n, v, tau, c, ldc, work )
call zlarfx ( side, m, n, v, tau, c, ldc, work )
```

Description

The routine `?larfx` applies a real/complex elementary reflector *H* to a real/complex *m*-by-*n* matrix *C*, from either the left or the right.
H is represented in the form
$H = I - tau * v * v'$, where *tau* is a real/complex scalar and *v* is a real/complex vector.

If *tau* = 0, then *H* is taken to be the unit matrix

Input Parameters

side — `CHARACTER*1`.
If *side* = 'L': form *H**C*
If *side* = 'R': form *C**H*.

m — `INTEGER`. The number of rows of the matrix *C*.

n — `INTEGER`. The number of columns of the matrix *C*.

v — `REAL` for `slarfx`
`DOUBLE PRECISION` for `dlarfx`
`COMPLEX` for `clarfx`
`COMPLEX*16` for `zlarfx`
Array, `DIMENSION`
(*m*) if *side* = 'L' or
(*n*) if *side* = 'R'.
The vector *v* in the representation of *H*.

tau	REAL for slarfx DOUBLE PRECISION for dlarfx COMPLEX for clarfx COMPLEX*16 for zlarfx The value *tau* in the representation of *H*.
c	REAL for slarfx DOUBLE PRECISION for dlarfx COMPLEX for clarfx COMPLEX*16 for zlarfx Array, DIMENSION (*ldc*,*n*). On entry, the *m*-by-*n* matrix C.
ldc	INTEGER. The leading dimension of the array *c*. *lda* ≥ (1,*m*).
work	REAL for slarfx DOUBLE PRECISION for dlarfx COMPLEX for clarfx COMPLEX*16 for zlarfx Workspace array, DIMENSION (*n*) if *side* = 'L' or (*m*) if *side* = 'R'. *work* is not referenced if *H* has order < 11.

Output Parameters

c	On exit, C is overwritten by the matrix *H**C* if *side* = 'L', or *C**H* if *side* = 'R'.

?largv

Generates a vector of plane rotations with real cosines and real/complex sines.

Syntax

```
call slargv ( n, x, incx, y, incy, c, incc )
call dlargv ( n, x, incx, y, incy, c, incc )
call clargv ( n, x, incx, y, incy, c, incc )
call zlargv ( n, x, incx, y, incy, c, incc )
```

Description

The routine generates a vector of real/complex plane rotations with real cosines, determined by elements of the real/complex vectors x and y.

For `slargv/dlargv`:

$$\begin{bmatrix} c(i) & s(i) \\ -s(i) & c(i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} a_i \\ 0 \end{bmatrix}, \text{ for } i = 1,2,\ldots,n$$

For `clargv/zlargv`:

$$\begin{bmatrix} c(i) & s(i) \\ -\mathrm{conjg}(s(i)) & c(i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} r_i \\ 0 \end{bmatrix}, \text{ for } i = 1,2,\ldots,n$$

where $c(i)^2 + \mathrm{abs}(s(i))^2 = 1$ and the following conventions are used (these are the same as in `clartg/zlartg` but differ from the BLAS Level 1 routine `crotg/zrotg`):
If $y_i = 0$, then $c(i) = 1$ and $s(i) = 0$;
If $x_i = 0$, then $c(i) = 0$ and $s(i)$ is chosen so that r_i is real.

Input Parameters

n — `INTEGER`. The number of plane rotations to be generated.

x, y — `REAL` for `slargv`
`DOUBLE PRECISION` for `dlargv`
`COMPLEX` for `clargv`
`COMPLEX*16` for `zlargv`
Arrays, `DIMENSION` (1+(*n*-1)**incx*) and (1+(*n*-1)**incy*), respectively.
On entry, the vectors x and y.

incx — `INTEGER`. The increment between elements of *x*.
incx > 0.

incy — `INTEGER`. The increment between elements of *y*.
incy > 0.

incc — `INTEGER`. The increment between elements of the output array *c*. *incc* > 0.

Output Parameters

x On exit, *x*(*i*) is overwritten by a_i (for real flavors), or by r_i (for complex flavors), for $i = 1,...,n$.

y On exit, the sines *s*(*i*) of the plane rotations.

c `REAL` for `slargv/clargv`
`DOUBLE PRECISION` for `dlargv/zlargv`
Array, `DIMENSION` (1+(*n*-1)**incc*). The cosines of the plane rotations.

?larnv

Returns a vector of random numbers from a uniform or normal distribution.

Syntax

```
call slarnv ( idist, iseed, n, x )
call dlarnv ( idist, iseed, n, x )
call clarnv ( idist, iseed, n, x )
call zlarnv ( idist, iseed, n, x )
```

Description

The routine `?larnv` returns a vector of *n* random real/complex numbers from a uniform or normal distribution.

This routine calls the auxiliary routine `?laruv` to generate random real numbers from a uniform (0,1) distribution, in batches of up to 128 using vectorisable code. The Box-Muller method is used to transform numbers from a uniform to a normal distribution.

Input Parameters

idist `INTEGER`. Specifies the distribution of the random numbers:
for `slarnv` and `dlanrv`:
= 1: uniform (0,1)
= 2: uniform (-1,1)
= 3: normal (0,1).
for `clarnv` and `zlanrv`:
= 1: real and imaginary parts each uniform (0,1)

	= 2: real and imaginary parts each uniform (-1,1) = 3: real and imaginary parts each normal (0,1) = 4: uniformly distributed on the disc abs(*z*) < 1 = 5: uniformly distributed on the circle abs(*z*) = 1
iseed	`INTEGER`. Array, `DIMENSION` (4). On entry, the seed of the random number generator; the array elements must be between 0 and 4095, and *iseed*(4) must be odd.
n	`INTEGER`. The number of random numbers to be generated.

Output Parameters

x	`REAL` for `slarnv` `DOUBLE PRECISION` for `dlarnv` `COMPLEX` for `clarnv` `COMPLEX*16` for `zlarnv` Array, `DIMENSION` (*n*). The generated random numbers.
iseed	On exit, the seed is updated.

?larrb

Provides limited bisection to locate eigenvalues for more accuracy.

Syntax

```
call slarrb ( n, d, l, ld, lld, ifirst, ilast, sigma,
              reltol, w, wgap, werr, work, iwork, info )
call dlarrb ( n, d, l, ld, lld, ifirst, ilast, sigma,
              reltol, w, wgap, werr, work, iwork, info )
```

Description

Given the relatively robust representation(RRR) LDL^T, the routine does “limited” bisection to locate the eigenvalues of LDL^T, *w*(*ifirst*) through *w*(*ilast*), to more accuracy. Intervals [*left*, *right*] are maintained by storing their mid-points and semi-widths in the arrays *w* and *werr* respectively.

Input Parameters

n — `INTEGER`. The order of the matrix.

d — `REAL` for `slarrb`
`DOUBLE PRECISION` for `dlarrb`
Array, `DIMENSION` (*n*). The *n* diagonal elements of the diagonal matrix *D*.

l — `REAL` for `slarrb`
`DOUBLE PRECISION` for `dlarrb`
Array, `DIMENSION` (*n*-1). The *n*-1 subdiagonal elements of the unit bidiagonal matrix *L*.

ld — `REAL` for `slarrb`
`DOUBLE PRECISION` for `dlarrb`
Array, `DIMENSION` (*n*-1). The *n*-1 elements L_i*D_i.

lld — `REAL` for `slarrb`
`DOUBLE PRECISION` for `dlarrb`
Array, `DIMENSION` (*n*-1). The *n*-1 elements $L_i*L_i*D_i$.

ifirst — `INTEGER`. The index of the first eigenvalue in the cluster.

ilast — `INTEGER`. The index of the last eigenvalue in the cluster.

sigma — `REAL` for `slarrb`
`DOUBLE PRECISION` for `dlarrb`
The shift used to form LDL^T (see `?larrf`).

reltol — `REAL` for `slarrb`
`DOUBLE PRECISION` for `dlarrb`
The relative tolerance.

w — `REAL` for `slarrb`
`DOUBLE PRECISION` for `dlarrb`
Array, `DIMENSION` (*n*). On input, *w*(*ifirst*) through *w*(*ilast*) are estimates of the corresponding eigenvalues of LDL^T .

wgap — `REAL` for `slarrb`
`DOUBLE PRECISION` for `dlarrb`
Array, `DIMENSION` (*n*). The gaps between the eigenvalues of LDL^T .

werr — `REAL` for `slarrb`
`DOUBLE PRECISION` for `dlarrb`
Array, `DIMENSION` (*n*). On input, *werr*(*ifirst*) through *werr*(*ilast*) are the errors in the estimates *w*(*ifirst*) through *w*(*ilast*).

work REAL for slarrb
DOUBLE PRECISION for dlarrb
Workspace array. Note that this parameter is never used in the routine.

iwork INTEGER.
Workspace array, DIMENSION (2*n*).

Output Parameters

w On output these estimates of the eigenvalues are “refined”.

wgap Very small gaps are changed on output.

werr On output, “refined” errors in the estimates *w*(*ifirst*) through *w*(*ilast*).

info INTEGER.
Error flag. Note that this parameter is never set in the routine.

?larre

Given the tridiagonal matrix T, sets small off-diagonal elements to zero and for each unreduced block T_i, finds base representations and eigenvalues.

Syntax

```
call slarre ( n, d, e, tol, nsplit, isplit, m, w, woff,
              gersch, work, info )
call dlarre ( n, d, e, tol, nsplit, isplit, m, w, woff,
              gersch, work, info )
```

Description

Given the tridiagonal matrix *T*, the routine sets "small" off-diagonal elements to zero, and for each unreduced block T_i, it finds

- the numbers σ_i
- the base $T_i - \sigma_i I = L_i D_i L_i^T$ representations and
- eigenvalues of each $L_i D_i L_i^T$.

The representations and eigenvalues found are then used by `?stegr` to compute the eigenvectors of a symmetric tridiagonal matrix. Currently, the base representations are limited to being positive or negative definite, and the eigenvalues of the definite matrices are found by the *dqds* algorithm (subroutine `?lasq2`). As an added benefit, `?larre` also outputs the *n* Gerschgorin intervals for each $L_i D_i L_i^T$.

Input Parameters

n — `INTEGER`. The order of the matrix.

d — `REAL` for `slarre`
`DOUBLE PRECISION` for `dlarre`
Array, `DIMENSION` (*n*). On entry, the *n* diagonal elements of the tridiagonal matrix *T*.

e — `REAL` for `slarre`
`DOUBLE PRECISION` for `dlarre`
Array, `DIMENSION` (*n*). On entry, the (*n*-1) subdiagonal elements of the tridiagonal matrix *T*; *e*(*n*) need not be set.

tol — `REAL` for `slarre`
`DOUBLE PRECISION` for `dlarre`
The threshold for splitting. If on input $|e(i)| < tol$, then the matrix *T* is split into smaller blocks.

nsplit — `INTEGER`. The number of blocks *T* splits into.
$1 \le nsplit \le n$.

work — `REAL` for `slarre`
`DOUBLE PRECISION` for `dlarre`
Workspace array, `DIMENSION` (4**n*).

Output Parameters

d — On exit, the *n* diagonal elements of the diagonal matrices D_i .

e — On exit, the subdiagonal elements of the unit bidiagonal matrices L_i .

isplit — `INTEGER`.
Array, `DIMENSION` (2*n*). The splitting points, at which *T* breaks up into submatrices. The first submatrix consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), etc., and the *nsplit*-th consists of rows/columns *isplit*(*nsplit*-1)+1 through *isplit*(*nsplit*)=*n*.

m	`INTEGER`. The total number of eigenvalues (of all the $L_i D_i L_i^T$) found.
w	`REAL` for `slarre` `DOUBLE PRECISION` for `dlarre` Array, `DIMENSION` (*n*). The first *m* elements contain the eigenvalues. The eigenvalues of each of the blocks, $L_i D_i L_i^T$, are sorted in ascending order.
woff	`REAL` for `slarre` `DOUBLE PRECISION` for `dlarre` Array, `DIMENSION` (*n*).The *nsplit* base points σ_i.
gersch	`REAL` for `slarre` `DOUBLE PRECISION` for `dlarre` Array, `DIMENSION` (2*n*). The *n* Gerschgorin intervals.
info	`INTEGER`. Output error code from `?lasq2`.

?larrf

Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.

Syntax

```
call slarrf ( n, d, l, ld, lld, ifirst, ilast, w, dplus,
              lplus, work, iwork, info )
call dlarrf ( n, d, l, ld, lld, ifirst, ilast, w, dplus,
              lplus, work, iwork, info )
```

Description

Given the initial representation LDL^T and its cluster of close eigenvalues (in a relative measure), *w*(*ifirst*), *w*(*ifirst*+1), ... *w*(*ilast*), the routine `?larrf` finds a new relatively robust representation

$$LDL^T - \sigma_i I = L(+)D(+)L(+)^T$$

such that at least one of the eigenvalues of $L(+)D(+)L(+)^T$ is relatively isolated.

Input Parameters

n	`INTEGER`. The order of the matrix.

d	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION (*n*). The *n* diagonal elements of the diagonal matrix *D*.
l	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION (*n*-1). The (*n*-1) subdiagonal elements of the unit bidiagonal matrix *L*.
ld	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION (*n*-1). The *n*-1 elements L_i*D_i.
lld	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION (*n*-1). The *n*-1 elements $L_i*L_i*D_i$.
ifirst	INTEGER. The index of the first eigenvalue in the cluster.
ilast	INTEGER. The index of the last eigenvalue in the cluster.
w	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION (*n*). On input, the eigenvalues of LDL^T in ascending order. *w*(*ifirst*) through *w*(*ilast*) form the cluster of relatively close eigenvalues.
sigma	REAL for slarrf DOUBLE PRECISION for dlarrf The shift used to form $L(+)D(+)L(+)^T$.
work	REAL for slarrf DOUBLE PRECISION for dlarrf Workspace array.

Output Parameters

w	On output, *w*(*ifirst*) through *w*(*ilast*) are estimates of the corresponding eigenvalues of $L(+)D(+)L(+)^T$.
dplus	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION (*n*). The *n* diagonal elements of the diagonal matrix *D*(+).

lplus REAL for `slarrf`
DOUBLE PRECISION for `dlarrf`
Array, DIMENSION (*n*). The first (*n*-1) elements of *lplus* contain the subdiagonal elements of the unit bidiagonal matrix *L*(+). *lplus*(*n*) is set to *sigma*.

?larrv

Computes the eigenvectors of the tridiagonal matrix T = L D L^T given L, D and the eigenvalues of L D L^T.

Syntax

```
call slarrv ( n, d, l, isplit, m, w, iblock, gersch,
              tol, z, ldz, isuppz, work, iwork, info )
call dlarrv ( n, d, l, isplit, m, w, iblock, gersch,
              tol, z, ldz, isuppz, work, iwork, info )
call clarrv ( n, d, l, isplit, m, w, iblock, gersch,
              tol, z, ldz, isuppz, work, iwork, info )
call zlarrv ( n, d, l, isplit, m, w, iblock, gersch,
              tol, z, ldz, isuppz, work, iwork, info )
```

Description

The routine `?larrv` computes the eigenvectors of the tridiagonal matrix $T = L D L^T$ given *L*, *D* and the eigenvalues of $L D L^T$. The input eigenvalues should have high relative accuracy with respect to the entries of *L* and *D*. The desired accuracy of the output can be specified by the input parameter *tol*.

Input Parameters

n INTEGER. The order of the matrix. $n \geq 0$.

d REAL for `slarrv/clarrv`
DOUBLE PRECISION for `dlarrv/zlarrv`
Array, DIMENSION (*n*). On entry, the *n* diagonal elements of the diagonal matrix *D*.

l	`REAL` for `slarrv/clarrv` `DOUBLE PRECISION` for `dlarrv/zlarrv` Array, `DIMENSION` (*n*-1). On entry, the (*n*-1) subdiagonal elements of the unit bidiagonal matrix *L* are contained in elements 1 to *n*-1 of *l*. *l*(*n*) need not be set.
isplit	`INTEGER`. Array, `DIMENSION` (*n*). The splitting points, at which *T* breaks up into submatrices. The first submatrix consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), etc.
tol	`REAL` for `slarrv/clarrv` `DOUBLE PRECISION` for `dlarrv/zlarrv` The absolute error tolerance for the eigenvalues/eigenvectors. Errors in the input eigenvalues must be bounded by *tol*. The eigenvectors output have residual norms bounded by *tol*, and the dot products between different eigenvectors are bounded by *tol*. *tol* must be at least *n***eps**|*T*|, where *eps* is the machine precision and |*T*| is the 1-norm of the tridiagonal matrix.
m	`INTEGER`. The total number of eigenvalues found. $0 \le m \le n$. If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu* - *il* +1.
w	`REAL` for `slarrv/clarrv` `DOUBLE PRECISION` for `dlarrv/zlarrv` Array, `DIMENSION` (*n*). The first *m* elements of *w* contain the eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block (The output array *w* from `?larre` is expected here). Errors in *w* must be bounded by *tol*.
iblock	`INTEGER`. Array, `DIMENSION` (*n*). The submatrix indices associated with the corresponding eigenvalues in *w*; *iblock*(*i*)=1 if eigenvalue *w*(*i*) belongs to the first submatrix from the top, =2 if *w*(*i*) belongs to the second submatrix, etc.
ldz	`INTEGER`. The leading dimension of the output array *z*. *ldz* ≥ 1, and if *jobz* = 'V', *ldz* ≥ max(1,*n*).
work	`REAL` for `slarrv/clarrv` `DOUBLE PRECISION` for `dlarrv/zlarrv` Workspace array, `DIMENSION` (13*n*).

iwork INTEGER.
Workspace array, DIMENSION (6*n*).

Output Parameters

d On exit, *d* may be overwritten.

l On exit, *l* is overwritten.

z REAL for slarrv
DOUBLE PRECISION for dlarrv
COMPLEX for clarrv
COMPLEX*16 for zlarrv
Array, DIMENSION (*ldz*, max(1,*m*)).
If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
If *jobz* = 'N', then *z* is not referenced.

NOTE. The user must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

isuppz INTEGER .
Array, DIMENSION (2*max(1,*m*)). The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th eigenvector is nonzero only in elements *isuppz*(2*i*-1) through *isuppz*(2*i*).

info INTEGER.
If *info* = 0: successful exit
If *info* = -*i* < 0: the *i*-th argument had an illegal value
info > 0: if *info* = 1, there is an internal error in ?larrb;
if *info* = 2, there is an internal error in ?stein.

?lartg

Generates a plane rotation with real cosine and real/complex sine.

Syntax

```
call slartg ( f, g, cs, sn, r )
call dlartg ( f, g, cs, sn, r )
call clartg ( f, g, cs, sn, r )
call zlartg ( f, g, cs, sn, r )
```

Description

The routine generates a plane rotation so that

$$\begin{bmatrix} cs & sn \\ -\mathrm{conjg}(sn) & cs \end{bmatrix} \cdot \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

where $cs^2 + |sn|^2 = 1$

This is a slower, more accurate version of the BLAS Level 1 routine `?rotg`, except for the following differences.

For `slartg/dlartg`:

f and *g* are unchanged on return;
If g=0, then cs=1 and sn=0;
If f=0 and $g \neq 0$, then cs=0 and sn=1 without doing any floating point operations (saves work in `?bdsqr` when there are zeros on the diagonal);
If *f* exceeds *g* in magnitude, *cs* will be positive.

For `clartg/zlartg`:

f and *g* are unchanged on return;
If g=0, then cs=1 and sn=0;
If f=0, then cs=0 and *sn* is chosen so that *r* is real.

Input Parameters

f, g — `REAL` for `slartg`
`DOUBLE PRECISION` for `dlartg`
`COMPLEX` for `clartg`
`COMPLEX*16` for `zlartg`
The first and second component of vector to be rotated.

Output Parameters

cs — `REAL` for `slartg/clartg`
`DOUBLE PRECISION` for `dlartg/zlartg`
The cosine of the rotation.

sn — `REAL` for `slartg`
`DOUBLE PRECISION` for `dlartg`
`COMPLEX` for `clartg`
`COMPLEX*16` for `zlartg`
The sine of the rotation.

r — `REAL` for `slartg`
`DOUBLE PRECISION` for `dlartg`
`COMPLEX` for `clartg`
`COMPLEX*16` for `zlartg`
The nonzero component of the rotated vector.

?lartv

Applies a vector of plane rotations with real cosines and real/complex sines to the elements of a pair of vectors.

Syntax

```
call slartv ( n, x, incx, y, incy, c, s, incc )
call dlartv ( n, x, incx, y, incy, c, s, incc )
call clartv ( n, x, incx, y, incy, c, s, incc )
call zlartv ( n, x, incx, y, incy, c, s, incc )
```

Description

The routine applies a vector of real/complex plane rotations with real cosines to elements of the real/complex vectors *x* and *y*. For i = 1,2,...,*n*

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} := \begin{bmatrix} c(i) & s(i) \\ -\mathrm{conjg}(s(i)) & c(i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

Input Parameters

n — `INTEGER`. The number of plane rotations to be applied.

x, y — `REAL` for `slartv`
`DOUBLE PRECISION` for `dlartv`
`COMPLEX` for `clartv`
`COMPLEX*16` for `zlartv`
Arrays, `DIMENSION` (1+(*n*-1)**incx*) and (1+(*n*-1)**incy*), respectively. The input vectors *x* and *y*.

incx — `INTEGER`. The increment between elements of *x*.
incx > 0.

incy — `INTEGER`. The increment between elements of *y*.
incy > 0.

c — `REAL` for `slartv`/`clartv`
`DOUBLE PRECISION` for `dlartv`/`zlartv`
Array, `DIMENSION` (1+(*n*-1)**incc*). The cosines of the plane rotations.

s — `REAL` for `slartv`
`DOUBLE PRECISION` for `dlartv`
`COMPLEX` for `clartv`
`COMPLEX*16` for `zlartv`
Array, `DIMENSION` (1+(*n*-1)**incc*). The sines of the plane rotations.

incc — `INTEGER`. The increment between elements of *c* and *s*. *incc* > 0.

Output Parameters

x, y — The rotated vectors *x* and *y*.

?laruv

Returns a vector of `n` *random real numbers from a uniform distribution.*

Syntax

```
call slaruv ( iseed, n, x )
call dlaruv ( iseed, n, x )
```

Description

The routine `?laruv` returns a vector of *n* random real numbers from a uniform (0,1) distribution ($n \le 128$).

This is an auxiliary routine called by `?larnv`.

Input Parameters

iseed	`INTEGER`. Array, `DIMENSION` (4). On entry, the seed of the random number generator; the array elements must be between 0 and 4095, and *iseed*(4) must be odd.
n	`INTEGER`. The number of random numbers to be generated. $n \le 128$.

Output Parameters

x	`REAL` for `slaruv` `DOUBLE PRECISION` for `dlaruv` Array, `DIMENSION` (*n*). The generated random numbers.
seed	On exit, the seed is updated.

?larz

Applies an elementary reflector (as returned by `?tzrzf`*) to a general matrix.*

Syntax

```
call slarz ( side, m, n, l, v, incv, tau, c, ldc, work )
call dlarz ( side, m, n, l, v, incv, tau, c, ldc, work )
call clarz ( side, m, n, l, v, incv, tau, c, ldc, work )
call zlarz ( side, m, n, l, v, incv, tau, c, ldc, work )
```

Description

The routine `?larz` applies a real/complex elementary reflector *H* to a real/complex *m*-by-*n* matrix *C*, from either the left or the right.
H is represented in the form
$H = I - tau * v * v'$,
where *tau* is a real/complex scalar and *v* is a real/complex vector.
If *tau* = 0, then *H* is taken to be the unit matrix.
For complex flavors, to apply *H'* (the conjugate transpose of *H*), supply conjg(*tau*) instead of *tau*.
H is a product of *k* elementary reflectors as returned by `?tzrzf`.

Input Parameters

side	`CHARACTER*1`. If *side* = 'L': form $H*C$ If *side* = 'R': form $C*H$
m	`INTEGER`. The number of rows of the matrix *C*.
n	`INTEGER`. The number of columns of the matrix *C*.
l	`INTEGER`. The number of entries of the vector *v* containing the meaningful part of the Householder vectors. If *side* = 'L', $m \ge l \ge 0$, if *side* = 'R', $n \ge l \ge 0$.
v	`REAL` for `slarz` `DOUBLE PRECISION` for `dlarz` `COMPLEX` for `clarz` `COMPLEX*16` for `zlarz`

	Array, DIMENSION (1+(*l*-1)*abs(*incv*)). The vector *v* in the representation of *H* as returned by ?tzrzf. *v* is not used if *tau* = 0.
incv	INTEGER. The increment between elements of *v*. *incv* ≠ 0.
tau	REAL for slarz DOUBLE PRECISION for dlarz COMPLEX for clarz COMPLEX*16 for zlarz The value *tau* in the representation of *H*.
c	REAL for slarz DOUBLE PRECISION for dlarz COMPLEX for clarz COMPLEX*16 for zlarz Array, DIMENSION (*ldc*,*n*). On entry, the *m*-by-*n* matrix *C*.
ldc	INTEGER. The leading dimension of the array *c*. *ldc* ≥ max(1,*m*).
work	REAL for slarz DOUBLE PRECISION for dlarz COMPLEX for clarz COMPLEX*16 for zlarz Workspace array, DIMENSION (*n*) if *side* = 'L' or (*m*) if *side* = 'R'.

Output Parameters

c	On exit, *c* is overwritten by the matrix *H***C* if *side* = 'L', or *C***H* if *side* = 'R'.

?larzb

Applies a block reflector or its transpose/conjugate-transpose to a general matrix.

```
call slarzb ( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc, work, ldwork )
call dlarzb ( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc, work, ldwork )
call clarzb ( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc, work, ldwork )
call zlarzb ( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc, work, ldwork )
```

Description

The routine applies a real/complex block reflector H or its transpose H^T (or H^H for complex flavors) to a real/complex distributed m-by-n matrix C from the left or the right.
Currently, only *storev* = 'R' and *direct* = 'B' are supported.

Input Parameters

side — CHARACTER*1.
If *side* = 'L': apply H or H' from the left
If *side* = 'R': apply H or H' from the right

trans — CHARACTER*1.
If *trans* = 'N': apply H (No transpose)
If *trans*='C': apply H' (Transpose/conjugate transpose)

direct — CHARACTER*1. Indicates how H is formed from a product of elementary reflectors
= 'F': $H = H(1) H(2)... H(k)$ (forward, not supported yet)
= 'B': $H = H(k)... H(2) H(1)$ (backward)

storev — CHARACTER*1. Indicates how the vectors which define the elementary reflectors are stored:
= 'C': Column-wise (not supported yet)
= 'R': Row-wise.

m — INTEGER. The number of rows of the matrix C.

n	`INTEGER`. The number of columns of the matrix *C*.
k	`INTEGER`. The order of the matrix *T* (equal to the number of elementary reflectors whose product defines the block reflector).
l	`INTEGER`. The number of columns of the matrix *V* containing the meaningful part of the Householder reflectors. If *side* = 'L', $m \geq l \geq 0$, if *side* = 'R', $n \geq l \geq 0$.
v	`REAL` for `slarzb` `DOUBLE PRECISION` for `dlarzb` `COMPLEX` for `clarzb` `COMPLEX*16` for `zlarzb` Array, `DIMENSION` (*ldv*, nv). If *storev* = 'C', nv = *k*; if *storev* = 'R', nv = *l*.
ldv	`INTEGER`. The leading dimension of the array *v*. If *storev* = 'C', $ldv \geq l$; if *storev* = 'R', $ldv \geq k$.
t	`REAL` for `slarzb` `DOUBLE PRECISION` for `dlarzb` `COMPLEX` for `clarzb` `COMPLEX*16` for `zlarzb` Array, `DIMENSION` (*ldt*,*k*). The triangular *k*-by-*k* matrix *T* in the representation of the block reflector.
ldt	`INTEGER`. The leading dimension of the array *t*. $ldt \geq k$.
c	`REAL` for `slarzb` `DOUBLE PRECISION` for `dlarzb` `COMPLEX` for `clarzb` `COMPLEX*16` for `zlarzb` Array, `DIMENSION` (*ldc*,*n*). On entry, the *m*-by-*n* matrix *C*.
ldc	`INTEGER`. The leading dimension of the array *c*. $ldc \geq \max(1,m)$.
work	`REAL` for `slarzb` `DOUBLE PRECISION` for `dlarzb` `COMPLEX` for `clarzb` `COMPLEX*16` for `zlarzb` Workspace array, `DIMENSION` (*ldwork*, *k*).

ldwork — `INTEGER`. The leading dimension of the array *work*.
If *side* = 'L', *ldwork* ≥ max(1, *n*);
if *side* = 'R', *ldwork* ≥ max(1, *m*).

Output Parameters

c — On exit, *c* is overwritten by $H*C$ or $H'*C$ or $C*H$ or $C*H'$.

?larzt

Forms the triangular factor T of a block reflector H = I - VTV^H.

Syntax

```
call slarzt ( direct, storev, n, k, v, ldv, tau, t, ldt )
call dlarzt ( direct, storev, n, k, v, ldv, tau, t, ldt )
call clarzt ( direct, storev, n, k, v, ldv, tau, t, ldt )
call zlarzt ( direct, storev, n, k, v, ldv, tau, t, ldt )
```

Description

The routine forms the triangular factor T of a real/complex block reflector H of order > *n*, which is defined as a product of *k* elementary reflectors.
If *direct* = 'F', $H = H(1)\ H(2)\ ...\ H(k)$ and T is upper triangular.
If *direct* = 'B', $H = H(k)\ ...\ H(2)\ H(1)$ and T is lower triangular.
If *storev* = 'C', the vector which defines the elementary reflector $H(i)$ is stored in the *i*-th column of the array *v*, and
$H = I - V*T*V'$
If *storev* = 'R', the vector which defines the elementary reflector $H(i)$ is stored in the *i*-th row of the array *v*, and
$H = I - V'*T*V$
Currently, only *storev* = 'R' and *direct* = 'B' are supported.

Input Parameters

direct — CHARACTER*1. Specifies the order in which the elementary reflectors are multiplied to form the block reflector:
If *direct* = 'F': $H = H(1)\ H(2) \ldots H(k)$ (forward, not supported yet)
If *direct* = 'B': $H = H(k) \ldots H(2)\ H(1)$ (backward)

storev — CHARACTER*1. Specifies how the vectors which define the elementary reflectors are stored (see also *Application Notes* below):
If *storev* = 'C': column-wise (not supported yet)
If *storev* = 'R': row-wise

n — INTEGER. The order of the block reflector *H*. $n \geq 0$.

k — INTEGER. The order of the triangular factor *T* (equal to the number of elementary reflectors). $k \geq 1$.

v — REAL for slarzt
DOUBLE PRECISION for dlarzt
COMPLEX for clarzt
COMPLEX*16 for zlarzt
Array, DIMENSION
(*ldv*, *k*) if *storev* = 'C'
(*ldv*, *n*) if *storev* = 'R'
The matrix *V*.

ldv — INTEGER. The leading dimension of the array *v*.
If *storev* = 'C', $ldv \geq \max(1,n)$;
if *storev* = 'R', $ldv \geq k$.

tau — REAL for slarzt
DOUBLE PRECISION for dlarzt
COMPLEX for clarzt
COMPLEX*16 for zlarzt
Array, DIMENSION (*k*). *tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*).

ldt — INTEGER. The leading dimension of the output array *t*.
$ldt \geq k$.

Output Parameters

t — REAL for slarzt
DOUBLE PRECISION for dlarzt
COMPLEX for clarzt

`COMPLEX*16` for `zlarzt`
Array, `DIMENSION` (*`ldt`*,*`k`*). The *`k`*-by-*`k`* triangular factor *T* of the block reflector. If *`direct`* = '`F`', *T* is upper triangular; if *`direct`* = '`B`', *T* is lower triangular. The rest of the array is not used.

`v` — The matrix *V*. See *Application Notes* below.

Application Notes

The shape of the matrix *V* and the storage of the vectors which define the *H*(i) is best illustrated by the following example with *n* = 5 and *k* = 3. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

`direct` = '`F`' and *`storev`* = '`C`':

$$V = \begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \end{bmatrix}$$

```
 .  .  .
 .  .  .
 1  .  .
    1  .
       1
```

`direct` = '`F`' and *`storev`* = '`R`':

```
 ____V___
/        \
```

$$\begin{bmatrix} v_1 & v_1 & v_1 & v_1 & v_1 & . & . & . & . & 1 \\ v_2 & v_2 & v_2 & v_2 & v_2 & . & . & . & 1 & \\ v_3 & v_3 & v_3 & v_3 & v_3 & . & . & 1 & & \end{bmatrix}$$

`direct` = 'B' and `storev` = 'C':   `direct` = 'B' and `storev` = 'R':

$$
\begin{matrix} 1 & & \\ . & 1 & \\ . & . & 1 \\ . & . & . \\ . & . & . \end{matrix}
$$

$$
V = \begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \end{bmatrix}
$$

$$
\overbrace{\qquad V \qquad}
$$

$$
\begin{bmatrix} 1 & . & . & . & . & v_1 & v_1 & v_1 & v_1 & v_1 \\ . & 1 & . & . & . & v_2 & v_2 & v_2 & v_2 & v_2 \\ . & . & 1 & . & . & v_3 & v_3 & v_3 & v_3 & v_3 \end{bmatrix}
$$

?las2

Computes singular values of a 2-by-2 triangular matrix.

Syntax

```
call slas2 ( f, g, h, ssmin, ssmax )
call dlas2 ( f, g, h, ssmin, ssmax )
```

Description

The routine `?las2` computes the singular values of the 2-by-2 matrix

$$
\begin{bmatrix} f & g \\ 0 & h \end{bmatrix}
$$

On return, *ssmin* is the smaller singular value and *ssmax* is the larger singular value.

Input Parameters

f, g, h	`REAL` for `slas2` `DOUBLE PRECISION` for `dlas2` The (1,1), (1,2) and (2,2) elements of the 2-by-2 matrix, respectively.

Output Parameters

ssmin, ssmax	`REAL` for `slas2` `DOUBLE PRECISION` for `dlas2` The smaller and the larger singular values, respectively.

Application Notes

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 correctly if one matrix element is infinite.
Overflow will not occur unless the largest singular value itself overflows, or is within a few ulps of overflow. (On machines with partial overflow, like the Cray, overflow may occur if the largest singular value is within a factor of 2 of overflow.)
Underflow is harmless if underflow is gradual. Otherwise, results may correspond to a matrix modified by perturbations of size near the underflow threshold.

?lascl

Multiplies a general rectangular matrix by a real scalar defined as c_{to}/c_{from}.

Syntax

```
call slascl ( type, kl, ku, cfrom, cto, m, n, a, lda, info )
call dlascl ( type, kl, ku, cfrom, cto, m, n, a, lda, info )
call clascl ( type, kl, ku, cfrom, cto, m, n, a, lda, info )
call zlascl ( type, kl, ku, cfrom, cto, m, n, a, lda, info )
```

Description

The routine `?lascl` multiplies the *m*-by-*n* real/complex matrix *A* by the real scalar *cto*/*cfrom*. The operation is performed without over/underflow as long as the final result *cto***A*(*i,j*)/*cfrom* does not over/underflow.

type specifies that *A* may be full, upper triangular, lower triangular, upper Hessenberg, or banded.

Input Parameters

type — `CHARACTER*1`. *type* indices the storage *type* of the input matrix.
= 'G': *A* is a full matrix.
= 'L': *A* is a lower triangular matrix.
= 'U': *A* is an upper triangular matrix.
= 'H': *A* is an upper Hessenberg matrix.
= 'B': *A* is a symmetric band matrix with lower bandwidth *kl* and upper bandwidth *ku* and with the only the lower half stored
= 'Q': *A* is a symmetric band matrix with lower bandwidth *kl* and upper bandwidth *ku* and with the only the upper half stored.
= 'Z': *A* is a band matrix with lower bandwidth *kl* and upper bandwidth *ku*.

kl — `INTEGER`. The lower bandwidth of *A*. Referenced only if *type* = 'B', 'Q' or 'Z'.

ku — `INTEGER`. The upper bandwidth of *A*. Referenced only if *type* = 'B', 'Q' or 'Z'.

cfrom, cto — `REAL` for `slascl/clascl`
`DOUBLE PRECISION` for `dlascl/zlascl`

The matrix *A* is multiplied by *cto*/*cfrom*. *A*(*i,j*) is computed without over/underflow if the final result *cto***A*(*i,j*)/*cfrom* can be represented without over/underflow. *cfrom* must be nonzero.

m — `INTEGER`. The number of rows of the matrix *A*. $m \geq 0$.

n — `INTEGER`. The number of columns of the matrix *A*.
$n \geq 0$.

a — `REAL` for `slascl`
`DOUBLE PRECISION` for `dlascl`
`COMPLEX` for `clascl`
`COMPLEX*16` for `zlascl`
Array, `DIMENSION` (*lda*, *m*). The matrix to be multiplied by *cto*/*cfrom*. See *type* for the storage type.

lda — `INTEGER`. The leading dimension of the array *a*.
$lda \geq \max(1,m)$.

Output Parameters

a The multiplied matrix *A*.

info INTEGER.
If *info* = 0 - successful exit
If *info* = -*i* < 0, the *i*-th argument had an illegal value.

?lasd0

Computes the singular values of a real upper bidiagonal n-by-m matrix B with diagonal d and off-diagonal e.
Used by ?bdsdc.

Syntax

```
call slasd0 ( n, sqre, d, e, u, ldu, vt, ldvt, smlsiz,
              iwork, work, info )
call dlasd0 ( n, sqre, d, e, u, ldu, vt, ldvt, smlsiz,
              iwork, work, info )
```

Description

Using a divide and conquer approach, the routine ?lasd0 computes the singular value decomposition (SVD) of a real upper bidiagonal *n*-by-*m* matrix *B* with diagonal *d* and offdiagonal *e*, where $m = n + sqre$.

The algorithm computes orthogonal matrices *U* and *VT* such that $B = U*S*VT$. The singular values *S* are overwritten on *d*.

A related subroutine, ?lasda, computes only the singular values, and optionally, the singular vectors in compact form.

Input Parameters

n INTEGER. On entry, the row dimension of the upper bidiagonal matrix. This is also the dimension of the main diagonal array *d*.

sqre INTEGER. Specifies the column dimension of the bidiagonal matrix.
If *sqre* = 0: The bidiagonal matrix has column dimension $m = n$;
If *sqre* = 1: The bidiagonal matrix has column dimension $m = n+1$;

d
REAL for slasd0
DOUBLE PRECISION for dlasd0
Array, DIMENSION (*n*). On entry, *d* contains the main diagonal of the bidiagonal matrix.

e
REAL for slasd0
DOUBLE PRECISION for dlasd0
Array, DIMENSION (*m*-1). Contains the subdiagonal entries of the bidiagonal matrix. On exit, *e* is destroyed.

ldu
INTEGER. On entry, leading dimension of the output array *u*.

ldvt
INTEGER. On entry, leading dimension of the output array *vt*.

smlsiz
INTEGER. On entry, maximum size of the subproblems at the bottom of the computation tree.

iwork
INTEGER.
Workspace array, DIMENSION must be at least (8*n*).

work
REAL for slasd0
DOUBLE PRECISION for dlasd0
Workspace array, DIMENSION must be at least
$(3m^2 + 2m)$.

Output Parameters

d
On exit *d*, if *info* = 0, contains singular values of the bidiagonal matrix.

u
REAL for slasd0
DOUBLE PRECISION for dlasd0
Array, DIMENSION at least (*ldq*, *n*). On exit, *u* contains the left singular vectors.

vt
REAL for slasd0
DOUBLE PRECISION for dlasd0
Array, DIMENSION at least (*ldvt*, *m*). On exit, *vt*' contains the right singular vectors.

info
INTEGER.
If *info* = 0: successful exit.
If *info* = -*i* < 0, the *i*-th argument had an illegal value.
If *info* = 1, an singular value did not converge.

?lasd1

Computes the SVD of an upper bidiagonal matrix B of the specified size. Used by `?bdsdc`.

Syntax

```
call slasd1 ( nl, nr, sqre, d, alpha, beta, u, ldu, vt,
              ldvt, idxq, iwork, work, info )
call dlasd1 ( nl, nr, sqre, d, alpha, beta, u, ldu, vt,
              ldvt, idxq, iwork, work, info )
```

Description

This routine computes the SVD of an upper bidiagonal n-by-m matrix B, where $n = nl + nr + 1$ and $m = n + sqre$. The routine `?lasd1` is called from `?lasd0`.

A related subroutine `?lasd7` handles the case in which the singular values (and the singular vectors in factored form) are desired.
`?lasd1` computes the SVD as follows:

$$B = U(in)*\begin{bmatrix} D1(in) & 0 & 0 & 0 \\ Z1' & a & Z2' & b \\ 0 & 0 & D2(in) & 0 \end{bmatrix}*VT(in)$$

$$= U(out)*(D(out)\ \ 0)*VT(out)$$

where $Z' = (Z1'\ a\ \ Z2'\ \ b) = u'\ \ VT'$, and u is a vector of dimension m with *alpha* and *beta* in the nl+1 and nl+2 -th entries and zeros elsewhere; and the entry b is empty if $sqre = 0$.

The left singular vectors of the original matrix are stored in *u*, and the transpose of the right singular vectors are stored in *vt*, and the singular values are in *d*. The algorithm consists of three stages:

The first stage consists of deflating the size of the problem when there are multiple singular values or when there are zeros in the Z vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine `?lasd2`.

The second stage consists of calculating the updated singular values. This is done by finding the square roots of the roots of the secular equation via the routine `?lasd4` (as called by `?lasd3`). This routine also calculates the singular vectors of the current problem.

The final stage consists of computing the updated singular vectors directly using the updated singular values. The singular vectors for the current problem are multiplied with the singular vectors from the overall problem.

Input Parameters

nl — `INTEGER`. The row dimension of the upper block.
$nl \geq 1$.

nr — `INTEGER`. The row dimension of the lower block.
$nr \geq 1$.

sqre — `INTEGER`.
If *sqre* = 0: the lower block is an *nr*-by-*nr* square matrix.
If *sqre* = 1: the lower block is an *nr*-by-(*nr*+1) rectangular matrix. The bidiagonal matrix has row dimension $n = nl + nr + 1$, and column dimension $m = n + sqre$.

d — `REAL` for `slasd1`
`DOUBLE PRECISION` for `dlasd1`
Array, `DIMENSION` ($n = nl+nr+1$). On entry *d*(1:*nl*,1:*nl*) contains the singular values of the upper block; and *d*(*nl*+2:*n*) contains the singular values of the lower block.

alpha — `REAL` for `slasd1`
`DOUBLE PRECISION` for `dlasd1`
Contains the diagonal element associated with the added row.

beta — `REAL` for `slasd1`
`DOUBLE PRECISION` for `dlasd1`
Contains the off-diagonal element associated with the added row.

u — `REAL` for `slasd1`
`DOUBLE PRECISION` for `dlasd1`
Array, `DIMENSION` (*ldu*, *n*). On entry *u*(1:*nl*, 1:*nl*) contains the left singular vectors of the upper block; *u*(*nl*+2:*n*, *nl*+2:*n*) contains the left singular vectors of the lower block.

ldu — `INTEGER`. The leading dimension of the array *u*.
$ldu \geq \max(1, n)$.

vt

REAL for slasd1
DOUBLE PRECISION for dlasd1
Array, DIMENSION (*ldvt*, *m*), where *m* = *n* + *sqre*.
On entry *vt*(1:*nl*+1, 1:*nl*+1)' contains the right singular vectors of the upper block; *vt*(*nl*+2:*m*, *nl*+2:*m*)' contains the right singular vectors of the lower block.

ldvt

INTEGER. The leading dimension of the array *vt*.
ldvt ≥ max(1, *m*).

iwork

INTEGER.
Workspace array, DIMENSION (4*n*).

work

REAL for slasd1
DOUBLE PRECISION for dlasd1
Workspace array, DIMENSION ($3m^2 + 2m$).

Output Parameters

d

On exit *d*(1:*n*) contains the singular values of the modified matrix.

u

On exit *u* contains the left singular vectors of the bidiagonal matrix.

vt

On exit *vt*' contains the right singular vectors of the bidiagonal matrix.

idxq

INTEGER
Array, DIMENSION (*n*). Contains the permutation which will reintegrate the subproblem just solved back into sorted order, that is, *d*(*idxq*(*i* = 1, *n*)) will be in ascending order.

info

INTEGER.
If *info* = 0: successful exit.
If *info* = -*i* < 0, the *i*-th argument had an illegal value.
If *info* = 1, an singular value did not converge.

?lasd2

Merges the two sets of singular values together into a single sorted set.
Used by `?bdsdc`.

Syntax

```
call slasd2 ( nl, nr, sqre, k, d, z, alpha, beta, u, ldu,
             vt, ldvt, dsigma, u2, ldu2, vt2, ldvt2,
              idxp, idx, idxc, idxq, coltyp, info )
call dlasd2 ( nl, nr, sqre, k, d, z, alpha, beta, u, ldu,
             vt, ldvt, dsigma, u2, ldu2, vt2, ldvt2,
              idxp, idx, idxc, idxq, coltyp, info )
```

Description

The routine `?lasd2` merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more singular values are close together or if there is a tiny entry in the *Z* vector. For each such occurrence the order of the related secular equation problem is reduced by one.

The routine `?lasd2` is called from `?lasd1`.

Input Parameters

nl — `INTEGER`. The row dimension of the upper block.
$nl \geq 1$.

nr — `INTEGER`. The row dimension of the lower block.
$nr \geq 1$.

sqre — `INTEGER`.
If $sqre = 0$: the lower block is an *nr*-by-*nr* square matrix
If $sqre = 1$: the lower block is an *nr*-by-(*nr*+1) rectangular matrix. The bidiagonal matrix has $n = nl + nr + 1$ rows and $m = n + sqre \geq n$ columns.

d — `REAL` for `slasd2`
`DOUBLE PRECISION` for `dlasd2`
Array, `DIMENSION` (*n*). On entry *d* contains the singular values of the two submatrices to be combined.

alpha REAL for slasd2
DOUBLE PRECISION for dlasd2
Contains the diagonal element associated with the added row.

beta REAL for slasd2
DOUBLE PRECISION for dlasd2
Contains the off-diagonal element associated with the added row.

u REAL for slasd2
DOUBLE PRECISION for dlasd2
Array, DIMENSION (*ldu*, *n*). On entry *u* contains the left singular vectors of two submatrices in the two square blocks with corners at (1,1), (*nl*, *nl*), and (*nl*+2, *nl*+2), (*n*,*n*).

ldu INTEGER. The leading dimension of the array *u*.
ldu ≥ *n*.

ldu2 INTEGER. The leading dimension of the output array *u2*. *ldu2* ≥ *n*.

vt REAL for slasd2
DOUBLE PRECISION for dlasd2
Array, DIMENSION (*ldvt*, *m*). On entry *vt*' contains the right singular vectors of two submatrices in the two square blocks with corners at (1,1), (*nl*+1, *nl*+1), and (*nl*+2, *nl*+2), (*m*,*m*).

ldvt INTEGER. The leading dimension of the array *vt*.
ldvt ≥ *m*.

ldvt2 INTEGER. The leading dimension of the output array *vt2*. *ldvt2* ≥ *m*.

idxp INTEGER.
Workspace array, DIMENSION (*n*). This will contain the permutation used to place deflated values of *d* at the end of the array. On output *idxp*(2:*k*) points to the nondeflated *d*-values and *idxp*(*k*+1:*n*) points to the deflated singular values.

idx INTEGER.
Workspace array, DIMENSION (*n*). This will contain the permutation used to sort the contents of *d* into ascending order.

coltyp INTEGER.
Workspace array, DIMENSION (*n*). As workspace, this will contain a label which will indicate which of the following types a column in the *u2* matrix or a row in the *vt2* matrix is:
1 : non-zero in the upper half only

2 : non-zero in the lower half only
3 : dense
4 : deflated.

idxq `INTEGER`.
Array, `DIMENSION` (*n*). This contains the permutation which separately sorts the two sub-problems in *d* into ascending order. Note that entries in the first half of this permutation must first be moved one position backward; and entries in the second half must first have *nl*+1 added to their values.

Output Parameters

k `INTEGER`. Contains the dimension of the non-deflated matrix, This is the order of the related secular equation. $1 \le k \le n$.

d On exit *d* contains the trailing (*n*-*k*) updated singular values (those which were deflated) sorted into increasing order.

u On exit *u* contains the trailing (*n*-*k*) updated left singular vectors (those which were deflated) in its last *n*-*k* columns.

z `REAL` for `slasd2`
`DOUBLE PRECISION` for `dlasd2`
Array, `DIMENSION` (*n*). On exit *z* contains the updating row vector in the secular equation.

dsigma `REAL` for `slasd2`
`DOUBLE PRECISION` for `dlasd2`
Array, `DIMENSION` (*n*). Contains a copy of the diagonal elements (*k*-1 singular values and one zero) in the secular equation.

u2 `REAL` for `slasd2`
`DOUBLE PRECISION` for `dlasd2`
Array, `DIMENSION` (*ldu2*, *n*). Contains a copy of the first *k*-1 left singular vectors which will be used by `?lasd3` in a matrix multiply (`?gemm`) to solve for the new left singular vectors. *u2* is arranged into four blocks. The first block contains a column with 1 at *nl*+1 and zero everywhere else; the second block contains non-zero entries only at and above *nl*; the third contains non-zero entries only below *nl*+1; and the fourth is dense.

vt On exit *vt*' contains the trailing (*n*-*k*) updated right singular vectors (those which were deflated) in its last *n*-*k* columns. In case *sqre* =1, the last row of *vt* spans the right null space.

vt2 REAL for slasd2
DOUBLE PRECISION for dlasd2
Array, DIMENSION (*ldvt2*, *n*). *vt2*' contains a copy of the first *k* right singular vectors which will be used by ?lasd3 in a matrix multiply (?gemm) to solve for the new right singular vectors. *vt2* is arranged into three blocks. The first block contains a row that corresponds to the special 0 diagonal element in *sigma*; the second block contains non-zeros only at and before *nl* +1; the third block contains non-zeros only at and after *nl* +2.

idxc INTEGER.
Array, DIMENSION (*n*). This will contain the permutation used to arrange the columns of the deflated *U* matrix into three groups: the first group contains non-zero entries only at and above *nl*, the second contains non-zero entries only below *nl*+2, and the third is dense.

coltyp On exit, it is an array of dimension 4, with *coltyp*(*i*) being the dimension of the *i*-th type columns.

info INTEGER.
If *info* = 0: successful exit
If *info* = -*i* < 0, the *i*-th argument had an illegal value.

?lasd3

Finds all square roots of the roots of the secular equation, as defined by the values in D and Z, and then updates the singular vectors by matrix multiplication. Used by ?bdsdc.

Syntax

```
call slasd3 ( nl, nr, sqre, k, d, q, ldq, dsigma, u, ldu,
              u2, ldu2, vt, ldvt, vt2, ldvt2, idxc, ctot,
              z, info )
call dlasd3 ( nl, nr, sqre, k, d, q, ldq, dsigma, u, ldu,
              u2, ldu2, vt, ldvt, vt2, ldvt2, idxc, ctot,
              z, info )
```

Description

The routine `?lasd3` finds all the square roots of the roots of the secular equation, as defined by the values in *D* and *Z*. It makes the appropriate calls to `?lasd4` and then updates the singular vectors by matrix multiplication.

The routine `?lasd3` is called from `?lasd1`.

Input Parameters

nl	`INTEGER`. The row dimension of the upper block. $nl \geq 1$.
nr	`INTEGER`. The row dimension of the lower block. $nr \geq 1$.
sqre	`INTEGER`. If $sqre = 0$: the lower block is an *nr*-by-*nr* square matrix. If $sqre = 1$: the lower block is an *nr*-by-(*nr*+1) rectangular matrix. The bidiagonal matrix has $n = nl + nr + 1$ rows and $m = n + sqre \geq n$ columns.
k	`INTEGER`.The size of the secular equation, $1 \leq k \leq n$.
q	`REAL` for `slasd3` `DOUBLE PRECISION` for `dlasd3` Workspace array, `DIMENSION` at least (*ldq*, *k*).
ldq	`INTEGER`. The leading dimension of the array *q*. $ldq \geq k$.
dsigma	`REAL` for `slasd3` `DOUBLE PRECISION` for `dlasd3` Array, `DIMENSION` (*k*). The first *k* elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.
u	`REAL` for `slasd3` `DOUBLE PRECISION` for `dlasd3` Array, `DIMENSION` (*ldu*, *n*). The last $n - k$ columns of this matrix contain the deflated left singular vectors.
ldu	`INTEGER`. The leading dimension of the array *u*. $ldu \geq n$.

u2 — `REAL` for `slasd3`
`DOUBLE PRECISION` for `dlasd3`
Array, `DIMENSION` (*ldu2*, *n*). The first *k* columns of this matrix contain the non-deflated left singular vectors for the split problem.

ldu2 — `INTEGER`. The leading dimension of the array *u2*.
ldu2 ≥ *n*.

vt — `REAL` for `slasd3`
`DOUBLE PRECISION` for `dlasd3`
Array, `DIMENSION` (*ldvt*, *m*). The last *m* - *k* columns of *vt*' contain the deflated right singular vectors.

ldvt — `INTEGER`. The leading dimension of the array *vt*.
ldvt ≥ *n*.

vt2 — `REAL` for `slasd3`
`DOUBLE PRECISION` for `dlasd3`
Array, `DIMENSION` (*ldvt2*, *n*). The first *k* columns of *vt2*' contain the non-deflated right singular vectors for the split problem.

ldvt2 — `INTEGER`. The leading dimension of the array *vt2*. *ldvt2* ≥ *n*.

idxc — `INTEGER`.
Array, `DIMENSION` (*n*). The permutation used to arrange the columns of *u* (and rows of *vt*) into three groups: the first group contains non-zero entries only at and above (or before) *nl* +1; the second contains non-zero entries only at and below (or after) *nl*+2; and the third is dense. The first column of *u* and the row of *vt* are treated separately, however. The rows of the singular vectors found by `?lasd4` must be likewise permuted before the matrix multiplies can take place.

ctot — `INTEGER`.
Array, `DIMENSION` (4). A count of the total number of the various types of columns in *u* (or rows in *vt*), as described in *idxc*. The fourth column type is any column which has been deflated.

z — `REAL` for `slasd3`
`DOUBLE PRECISION` for `dlasd3`
Array, `DIMENSION` (*k*). The first *k* elements of this array contain the components of the deflation-adjusted updating row vector.

Output Parameters

d — `REAL` for `slasd3`
`DOUBLE PRECISION` for `dlasd3`
Array, `DIMENSION` (*k*). On exit the square roots of the roots of the secular equation, in ascending order.

info — `INTEGER`.
If *info* = 0: successful exit.
If *info* = -*i* < 0, the *i*-th argument had an illegal value.
If *info* = 1, an singular value did not converge.

Application Notes

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray XMP, Cray YMP, Cray C 90, or Cray 2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

?lasd4

Computes the square root of the i*-th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix. Used by* `?bdsdc`.

Syntax

```
call slasd4 ( n, i, d, z, delta, rho, sigma, work, info )
call dlasd4 ( n, i, d, z, delta, rho, sigma, work, info )
```

Description

This routine computes the square root of the *i*-th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix whose entries are given as the squares of the corresponding entries in the array *d*, and that $0 \le d(i) < d(j)$ for $i < j$ and that *rho* > 0. This is arranged by the calling routine, and is no loss in generality. The rank-one modified system is thus

diag(*d*)* diag(*d*) + *rho* **Z* * *Z*_transpose

where we assume the Euclidean norm of *Z* is 1.The method consists of approximating the rational functions in the secular equation by simpler interpolating rational functions.

Input Parameters

n INTEGER. The length of all arrays.

i INTEGER. The index of the eigenvalue to be computed. $1 \le i \le n$.

d REAL for slasd4
DOUBLE PRECISION for dlasd4
Array, DIMENSION (*n*).
The original eigenvalues. It is assumed that they are in order, $0 \le d(i) < d(j)$ for $i < j$.

z REAL for slasd4
DOUBLE PRECISION for dlasd4
Array, DIMENSION (*n*).
The components of the updating vector.

rho REAL for slasd4
DOUBLE PRECISION for dlasd4
The scalar in the symmetric updating formula.

work REAL for slasd4
DOUBLE PRECISION for dlasd4
Workspace array, DIMENSION (*n*).
If $n \ne 1$, *work* contains (*d*(*j*) + *sigma_i*) in its *j*-th component. If $n = 1$, then *work*(1) = 1.

Output Parameters

delta REAL for slasd4
DOUBLE PRECISION for dlasd4
Array, DIMENSION (*n*).
If $n \ne 1$, *delta* contains (*d*(*j*) - *sigma_i*) in its *j*-th component. If $n = 1$, then *delta* (1) = 1. The vector *delta* contains the information necessary to construct the (singular) eigenvectors.

sigma REAL for slasd4
DOUBLE PRECISION for dlasd4
The computed λ_i, the *i*-th updated eigenvalue.

info INTEGER.
= 0: successful exit
> 0: if *info* = 1, the updating process failed.

?lasd5

Computes the square root of the `i`-th *eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix. Used by* `?bdsdc`.

Syntax

```
call slasd5 ( i, d, z, delta, rho, dsigma, work )
call dlasd5 ( i, d, z, delta, rho, dsigma, work )
```

Description

This routine computes the square root of the *i*-th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix

diag(*d*)* diag(*d*) + *rho* **Z* * *Z*_transpose

The diagonal entries in the array *d* are assumed to satisfy $0 \le d(i) < d(j)$ for $i < j$. We also assume *rho* > 0 and that the Euclidean norm of the vector *Z* is one.

Input Parameters

i — `INTEGER`. The index of the eigenvalue to be computed. *i* = 1 or *i* = 2.

d — `REAL` for `slasd5`
`DOUBLE PRECISION` for `dlasd5`
Array, `DIMENSION` (2).
The original eigenvalues. We assume $0 \le d(1) < d(2)$.

z — `REAL` for `slasd5`
`DOUBLE PRECISION` for `dlasd5`
Array, `DIMENSION` (2).
The components of the updating vector.

rho — `REAL` for `slasd5`
`DOUBLE PRECISION` for `dlasd5`
The scalar in the symmetric updating formula.

work — `REAL` for `slasd5`
`DOUBLE PRECISION` for `dlasd5`.
Workspace array, `DIMENSION` (2).
Contains (*d*(*j*) + *sigma_i*) in its *j*-th component.

Output Parameters

delta — `REAL` for `slasd5`
`DOUBLE PRECISION` for `dlasd5`.
Array, `DIMENSION` (2).
Contains $(d(j) - \lambda_i)$ in its j-th component. The vector *delta* contains the information necessary to construct the eigenvectors.

dsigma — `REAL` for `slasd5`
`DOUBLE PRECISION` for `dlasd5`.
The computed λ_i, the i-th updated eigenvalue.

?lasd6

Computes the SVD of an updated upper bidiagonal matrix obtained by merging two smaller ones by appending a row. Used by `?bdsdc`.

Syntax

```
call slasd6 ( icompq, nl, nr, sqre, d, vf, vl, alpha, beta, idxq, perm, givptr,
    givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, iwork,
    info)
call dlasd6 ( icompq, nl, nr, sqre, d, vf, vl, alpha, beta, idxq, perm, givptr,
    givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, iwork,
    info)
```

Description

The routine `?lasd6` computes the *SVD* of an updated upper bidiagonal matrix *B* obtained by merging two smaller ones by appending a row. This routine is used only for the problem which requires all singular values and optionally singular vector matrices in factored form. *B* is an n-by-m matrix with
$n = nl + nr + 1$ and $m = n + sqre$. *A* related subroutine, `?lasd1`, handles the case in which all singular values and singular vectors of the bidiagonal matrix are desired. `?lasd6` computes the *SVD* as follows:

$$B = U(in)*\begin{bmatrix} D1(in) & 0 & 0 & 0 \\ Z1' & a & Z2' & b \\ 0 & 0 & D2(in) & 0 \end{bmatrix}*VT(in)$$

$$= U(out)*(D(out)\ \ 0)*VT(out)$$

where $Z' = (Z1'\ a\ \ Z2'\ \ b) = u'\ \ VT'$, and u is a vector of dimension m with `alpha` and `beta` in the `nl`+1 and `nl`+2 -th entries and zeros elsewhere; and the entry b is empty if `sqre` = 0.

The singular values of *B* can be computed using *D1, D2*, the first components of all the right singular vectors of the lower block, and the last components of all the right singular vectors of the upper block. These components are stored and updated in `vf` and `vl`, respectively, in `?lasd6`. Hence *U* and *VT* are not explicitly referenced.
The singular values are stored in *D*. The algorithm consists of two stages:
the first stage consists of deflating the size of the problem when there are multiple singular values or if there is a zero in the *Z* vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine `?lasd7`.

The second stage consists of calculating the updated singular values. This is done by finding the roots of the secular equation via the routine `?lasd4` (as called by `?lasd8`). This routine also updates `vf` and `vl` and computes the distances between the updated singular values and the old singular values. `?lasd6` is called from `?lasda.`

Input Parameters

icompq `INTEGER`. Specifies whether singular vectors are to be computed in factored form:
= 0: Compute singular values only
= 1: Compute singular vectors in factored form as well.

nl `INTEGER`.The row dimension of the upper block.
$nl \geq 1$.

nr `INTEGER`.The row dimension of the lower block.
$nr \geq 1$.

sqre INTEGER.
= 0: the lower block is an *nr*-by-*nr* square matrix.
= 1: the lower block is an *nr*-by-(*nr*+1) rectangular matrix.
The bidiagonal matrix has row dimension *n*=*nl*+*nr*+1, and column dimension *m* = *n* + *sqre*.

d REAL for slasd6
DOUBLE PRECISION for dlasd6
Array, DIMENSION (*nl*+*nr*+1). On entry *d*(1:*nl*,1:*nl*) contains the singular values of the upper block, and *d*(*nl*+2:*n*) contains the singular values of the lower block.

vf REAL for slasd6
DOUBLE PRECISION for dlasd6
Array, DIMENSION (*m*). On entry, *vf*(1:*nl*+1) contains the first components of all right singular vectors of the upper block; and *vf*(*nl*+2:*m*) contains the first components of all right singular vectors of the lower block.

vl REAL for slasd6
DOUBLE PRECISION for dlasd6
Array, DIMENSION (*m*). On entry, *vl*(1:*nl*+1) contains the last components of all right singular vectors of the upper block; and *vl*(*nl*+2:*m*) contains the last components of all right singular vectors of the lower block.

alpha REAL for slasd6
DOUBLE PRECISION for dlasd6
Contains the diagonal element associated with the added row.

beta REAL for slasd6
DOUBLE PRECISION for dlasd6
Contains the off-diagonal element associated with the added row.

ldgcol INTEGER.The leading dimension of the output array *givcol*, must be at least *n*.

ldgnum INTEGER. The leading dimension of the output arrays *givnum* and *poles*, must be at least *n*.

work REAL for slasd6
DOUBLE PRECISION for dlasd6
Workspace array, DIMENSION (4*m*).

iwork INTEGER
Workspace array, DIMENSION (3*n*).

Output Parameters

d On exit *d*(1:*n*) contains the singular values of the modified matrix.

vf On exit, *vf* contains the first components of all right singular vectors of the bidiagonal matrix.

vl On exit, *vl* contains the last components of all right singular vectors of the bidiagonal matrix.

idxq INTEGER.
Array, DIMENSION (*n*). This contains the permutation which will reintegrate the subproblem just solved back into sorted order, that is, *d*(*idxq*(*i* = 1, *n*)) will be in ascending order.

perm INTEGER.
Array, DIMENSION (*n*). The permutations (from deflation and sorting) to be applied to each block. Not referenced if *icompq* = 0.

givptr INTEGER. The number of Givens rotations which took place in this subproblem. Not referenced if *icompq* = 0.

givcol INTEGER.
Array, DIMENSION (*ldgcol*, 2). Each pair of numbers indicates a pair of columns to take place in a Givens rotation. Not referenced if *icompq* = 0.

givnum REAL for slasd6
DOUBLE PRECISION for dlasd6
Array, DIMENSION (*ldgnum*, 2). Each number indicates the *C* or *S* value to be used in the corresponding Givens rotation. Not referenced if *icompq* = 0.

poles REAL for slasd6
DOUBLE PRECISION for dlasd6
Array, DIMENSION (*ldgnum*, 2). On exit, *poles*(1,*) is an array containing the new singular values obtained from solving the secular equation, and *poles*(2,*) is an array containing the poles in the secular equation. Not referenced if *icompq* = 0.

difl REAL for slasd6
DOUBLE PRECISION for dlasd6
Array, DIMENSION (*n*). On exit, *difl*(*i*) is the distance between *i*-th updated (undeflated) singular value and the *i*-th (undeflated) old singular value.

difr REAL for slasd6
DOUBLE PRECISION for dlasd6
Array,

DIMENSION (*ldgnum*, 2) if *icompq* = 1 and DIMENSION (*n*) if *icompq* = 0. On exit, *difr*(*i*, 1) is the distance between *i*-th updated (undeflated) singular value and the *i*+1-th (undeflated) old singular value. If *icompq* = 1, *difr*(1:*k*, 2) is an array containing the normalizing factors for the right singular vector matrix.

See ?lasd8 for details on *difl* and *difr*.

z — REAL for slasd6
DOUBLE PRECISION for dlasd6
Array, DIMENSION (*m*).
The first elements of this array contain the components of the deflation-adjusted updating row vector.

k — INTEGER. Contains the dimension of the non-deflated matrix. This is the order of the related secular equation. $1 \le k \le n$.

c — REAL for slasd6
DOUBLE PRECISION for dlasd6
c contains garbage if *sqre* =0 and the *C*-value of a Givens rotation related to the right null space if
sqre = 1.

s — REAL for slasd6
DOUBLE PRECISION for dlasd6
s contains garbage if *sqre* =0 and the *S*-value of a Givens rotation related to the right null space if
sqre = 1.

info — INTEGER.
= 0: successful exit.
< 0: if *info* = -*i*, the *i*-th argument had an illegal value.
>0:if *info* = 1, an singular value did not converge

?lasd7

Merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. Used by ?bdsdc.

Syntax

```
call slasd7 ( icompq, nl, nr, sqre, k, d, z, zw, vf, vfw, vl, vlw, alpha, beta,
    dsigma, idx, idxp, idxq, perm, givptr, givcol, ldgcol, givnum, ldgnum, c, s,
    info )
call dlasd7 ( icompq, nl, nr, sqre, k, d, z, zw, vf, vfw, vl, vlw, alpha, beta,
    dsigma, idx, idxp, idxq, perm, givptr, givcol, ldgcol, givnum, ldgnum, c, s,
    info )
```

Description

The routine ?lasd7 merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more singular values are close together or if there is a tiny entry in the Z vector. For each such occurrence the order of the related secular equation problem is reduced by one. ?lasd7 is called from ?lasd6.

Input Parameters

icompq INTEGER.Specifies whether singular vectors are to be computed in compact form, as follows:
= 0: Compute singular values only.
= 1: Compute singular vectors of upper bidiagonal matrix in compact form.

nl INTEGER. The row dimension of the upper block.
nl ≥ 1.

nr INTEGER. The row dimension of the lower block.
nr ≥ 1.

sqre INTEGER.
= 0: the lower block is an *nr*-by-*nr* square matrix.
= 1: the lower block is an *nr*-by-(*nr*+1) rectangular matrix. The bidiagonal matrix has $n = nl + nr + 1$ rows and $m = n + sqre \geq n$ columns.

d REAL for slasd7
DOUBLE PRECISION for dlasd7
Array, DIMENSION (*n*). On entry *d* contains the singular values of the two submatrices to be combined.

zw REAL for slasd7
DOUBLE PRECISION for dlasd7
Array, DIMENSION (*m*). Workspace for *z*.

vf REAL for slasd7
DOUBLE PRECISION for dlasd7
Array, DIMENSION (*m*). On entry, *vf*(1:*nl*+1) contains the first components of all right singular vectors of the upper block; and *vf*(*nl*+2:*m*) contains the first components of all right singular vectors of the lower block.

vfw REAL for slasd7
DOUBLE PRECISION for dlasd7
Array, DIMENSION (*m*). Workspace for *vf*.

vl REAL for slasd7
DOUBLE PRECISION for dlasd7
Array, DIMENSION (*m*). On entry, *vl*(1:*nl*+1) contains the last components of all right singular vectors of the upper block; and *vl*(*nl*+2:*m*) contains the last components of all right singular vectors of the lower block.

vlw REAL for slasd7
DOUBLE PRECISION for dlasd7
Array, DIMENSION (*m*). Workspace for *vl*.

alpha REAL for slasd7
DOUBLE PRECISION for dlasd7.
Contains the diagonal element associated with the added row.

beta REAL for slasd7
DOUBLE PRECISION for dlasd7
Contains the off-diagonal element associated with the added row.

idx INTEGER.
Workspace array, DIMENSION (*n*). This will contain the permutation used to sort the contents of *d* into ascending order.

idxp INTEGER.
Workspace array, DIMENSION (*n*). This will contain the permutation used to place deflated values of *d* at the end of the array.

idxq	INTEGER. Array, DIMENSION (*n*). This contains the permutation which separately sorts the two sub-problems in *d* into ascending order. Note that entries in the first half of this permutation must first be moved one position backward; and entries in the second half must first have *nl*+1 added to their values.
ldgcol	INTEGER.The leading dimension of the output array *givcol*, must be at least *n*.
ldgnum	INTEGER. The leading dimension of the output array *givnum*, must be at least *n*.

Output Parameters

k	INTEGER. Contains the dimension of the non-deflated matrix, this is the order of the related secular equation. $1 \le k \le n$.
d	On exit, *d* contains the trailing (*n*-*k*) updated singular values (those which were deflated) sorted into increasing order.
z	REAL for slasd7 DOUBLE PRECISION for dlasd7. Array, DIMENSION (*m*). On exit, *z* contains the updating row vector in the secular equation.
vf	On exit, *vf* contains the first components of all right singular vectors of the bidiagonal matrix.
vl	On exit, *vl* contains the last components of all right singular vectors of the bidiagonal matrix.
dsigma	REAL for slasd7 DOUBLE PRECISION for dlasd7. Array, DIMENSION (*n*). Contains a copy of the diagonal elements (*k*-1 singular values and one zero) in the secular equation.
idxp	On output, *idxp*(2:*k*) points to the nondeflated *d*-values and *idxp*(*k*+1:*n*) points to the deflated singular values.
perm	INTEGER. Array, DIMENSION (*n*). The permutations (from deflation and sorting) to be applied to each singular block. Not referenced if *icompq* = 0.
givptr	INTEGER.The number of Givens rotations which took place in this subproblem. Not referenced if *icompq* = 0.

givcol `INTEGER`.
Array, `DIMENSION` (*ldgcol*, 2). Each pair of numbers indicates a pair of columns to take place in a Givens rotation. Not referenced if *icompq* = 0.

givnum `REAL for slasd7`
`DOUBLE PRECISION for dlasd7`.
Array, `DIMENSION` (*ldgnum*, 2). Each number indicates the *C* or *S* value to be used in the corresponding Givens rotation. Not referenced if *icompq* = 0.

c `REAL for slasd7.`
`DOUBLE PRECISION for dlasd7.`
c contains garbage if *sqre* =0 and the *C*-value of a Givens rotation related to the right null space if
sqre = 1.

s `REAL for slasd7.`
`DOUBLE PRECISION for dlasd7.`
s contains garbage if *sqre* =0 and the *S*-value of a Givens rotation related to the right null space if
sqre = 1.

info `INTEGER`.
= 0: successful exit.
< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?lasd8

Finds the square roots of the roots of the secular equation, and stores, for each element in D, the distance to its two nearest poles. Used by `?bdsdc`.

Syntax

```
call slasd8 ( icompq, k, d, z, vf, vl, difl, difr, lddifr, dsigma, work, info )
call dlasd8 ( icompq, k, d, z, vf, vl, difl, difr, lddifr, dsigma, work, info )
```

Description

The routine `?lasd8` finds the square roots of the roots of the secular equation, as defined by the values in *dsigma* and *z*. It makes the appropriate calls to `?lasd4`, and stores, for each element in *d*, the distance to its two nearest poles (elements in *dsigma*). It also updates the arrays *vf* and *vl*, the first and last components of all the right singular vectors of the original bidiagonal matrix. `?lasd8` is called from `?lasd6`.

Input Parameters

icompq — `INTEGER`. Specifies whether singular vectors are to be computed in factored form in the calling routine:
= 0: Compute singular values only.
= 1: Compute singular vectors in factored form as well.

k — `INTEGER`. The number of terms in the rational function to be solved by `?lasd4`. $k \geq 1$.

z — `REAL` for `slasd8`
`DOUBLE PRECISION` for `dlasd8`.
Array, `DIMENSION` (*k*). The first *k* elements of this array contain the components of the deflation-adjusted updating row vector.

vf — `REAL` for `slasd8`
`DOUBLE PRECISION` for `dlasd8`.
Array, `DIMENSION` (*k*). On entry, *vf* contains information passed through `dbede8`.

vl — `REAL` for `slasd8`
`DOUBLE PRECISION` for `dlasd8`.
Array, `DIMENSION` (*k*).On entry, *vl* contains information passed through `dbede8`.

lddifr — `INTEGER`.The leading dimension of the output array *difr*, must be at least *k*.

dsigma — `REAL` for `slasd8`
`DOUBLE PRECISION` for `dlasd8`.
Array, `DIMENSION` (*k*). The first *k* elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.

work — `REAL` for `slasd8`
`DOUBLE PRECISION` for `dlasd8`.
Workspace array, `DIMENSION` at least (3*k*).

Output Parameters

d — REAL for slasd8
DOUBLE PRECISION for dlasd8.
Array, DIMENSION (*k*). On output, *d* contains the updated singular values.

vf — On exit, *vf* contains the first *k* components of the first components of all right singular vectors of the bidiagonal matrix.

vl — On exit, *vl* contains the first *k* components of the last components of all right singular vectors of the bidiagonal matrix.

difl — REAL for slasd8
DOUBLE PRECISION for dlasd8.
Array, DIMENSION (*k*). On exit, *difl*(*i*) = *d*(*i*) - *dsigma*(*i*).

difr — REAL for slasd8
DOUBLE PRECISION for dlasd8.
Array,
DIMENSION (*lddifr*, 2) if *icompq* = 1 and
DIMENSION (*k*) if *icompq* = 0.
On exit, *difr*(*i*,1) = *d*(*i*) - *dsigma*(*i*+1), *difr*(*k*,1) is not defined and will not be referenced.
If *icompq* = 1, *difr*(1:*k*,2) is an array containing the normalizing factors for the right singular vector matrix.

info — INTEGER.
= 0: successful exit.
< 0: if *info* = -*i*, the *i*-th argument had an illegal value. > 0: if *info* = 1, an singular value did not converge.

?lasd9

Finds the square roots of the roots of the secular equation, and stores, for each element in D, the distance to its two nearest poles. Used by ?bdsdc.

Syntax

```
call slasd9 ( icompq, ldu, k, d, z, vf, vl, difl, difr, dsigma, work, info )
call dlasd9 ( icompq, ldu, k, d, z, vf, vl, difl, difr, dsigma, work, info )
```

Description

The routine `?lasd9` finds the square roots of the roots of the secular equation, as defined by the values in *dsigma* and *z*. It makes the appropriate calls to `?lasd4`, and stores, for each element in *d*, the distance to its two nearest poles (elements in *dsigma*). It also updates the arrays *vf* and *vl*, the first and last components of all the right singular vectors of the original bidiagonal matrix. `?lasd9` is called from `?lasd7`.

Input Parameters

icompq — `INTEGER`.Specifies whether singular vectors are to be computed in factored form in the calling routine:
If *icompq* = 0, compute singular values only;
If *icompq* = 1, compute singular vector matrices in factored form also.

k — `INTEGER`.The number of terms in the rational function to be solved by `slasd4`. $k \geq 1$.

dsigma — `REAL` for `slasd9`
`DOUBLE PRECISION` for `dlasd9`.
Array, `DIMENSION`(*k*). The first *k* elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.

z — `REAL` for `slasd9`
`DOUBLE PRECISION` for `dlasd9`.
Array, `DIMENSION` (*k*). The first *k* elements of this array contain the components of the deflation-adjusted updating row vector.

vf — `REAL` for `slasd9`
`DOUBLE PRECISION` for `dlasd9`.
Array, `DIMENSION`(*k*). On entry, *vf* contains information passed through `sbede8`.

vl — `REAL` for `slasd9`
`DOUBLE PRECISION` for `dlasd9`.
Array, `DIMENSION`(*k*). On entry, *vl* contains information passed through `sbede8`.

work — `REAL` for `slasd9`
`DOUBLE PRECISION` for `dlasd9`.
Workspace array, `DIMENSION` at least (3*k*).

Output Parameters

d
: `REAL` for `slasd9`
`DOUBLE PRECISION` for `dlasd9`.
Array, `DIMENSION`(*k*). *d*(*i*) contains the updated singular values.

vf
: On exit, *vf* contains the first *k* components of the first components of all right singular vectors of the bidiagonal matrix.

vl
: On exit, *vl* contains the first *k* components of the last components of all right singular vectors of the bidiagonal matrix.

difl
: `REAL` for `slasd9`
`DOUBLE PRECISION` for `dlasd9`.
Array, `DIMENSION` (*k*).
On exit, *difl*(*i*) = *d*(*i*) - *dsigma*(*i*).

difr
: `REAL` for `slasd9`
`DOUBLE PRECISION` for `dlasd9`.
Array,
`DIMENSION` (*ldu*, 2) if *icompq* =1 and
`DIMENSION` (*k*) if *icompq* = 0.
On exit, *difr*(*i*, 1) = *d*(*i*) - *dsigma*(*i*+1), *difr*(*k*, 1) is not defined and will not be referenced.
If *icompq* = 1, *difr*(1:*k*, 2) is an array containing the normalizing factors for the right singular vector matrix.

info
: `INTEGER`.
= 0: successful exit.
< 0: if *info* = -*i*, the *i*-th argument had an illegal value.
> 0: if *info* = 1, an singular value did not converge

?lasda

Computes the singular value decomposition (SVD) of a real upper bidiagonal matrix with diagonal `d` *and off-diagonal* `e`*. Used by* `?bdsdc`.

Syntax

```
call slasda ( icompq, smlsiz, n, sqre, d, e, u, ldu, vt, k, difl, difr, z,
   poles, givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info )
```

```
call dlasda ( icompq, smlsiz, n, sqre, d, e, u, ldu, vt, k, difl, difr, z,
    poles, givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info )
```

Description

Using a divide and conquer approach, `?lasda` computes the singular value decomposition *(SVD)* of a real upper bidiagonal *n*-by-*m* matrix *B* with diagonal *d* and off-diagonal *e*, where $m = n + sqre$. The algorithm computes the singular values in the *SVD* $B = U*S*VT$. The orthogonal matrices *U* and *VT* are optionally computed in compact form. *A* related subroutine, `?lasd0`, computes the singular values and the singular vectors in explicit form.

Input Parameters

icompq	`INTEGER`. Specifies whether singular vectors are to be computed in compact form, as follows: = 0: Compute singular values only. = 1: Compute singular vectors of upper bidiagonal matrix in compact form.
smlsiz	`INTEGER`. The maximum size of the subproblems at the bottom of the computation tree.
n	`INTEGER`. The row dimension of the upper bidiagonal matrix. This is also the dimension of the main diagonal array *d*.
sqre	`INTEGER`. Specifies the column dimension of the bidiagonal matrix. If *sqre* = 0: The bidiagonal matrix has column dimension *m* = *n*; If *sqre* = 1: The bidiagonal matrix has column dimension *m* = *n* + 1.
d	`REAL` for `slasda` `DOUBLE PRECISION for dlasda`. Array, `DIMENSION` (*n*). On entry *d* contains the main diagonal of the bidiagonal matrix.
e	`REAL` for `slasda` `DOUBLE PRECISION` for `dlasda`. Array, `DIMENSION` (*m* -1). Contains the subdiagonal entries of the bidiagonal matrix. On exit, *e* has been destroyed.
ldu	`INTEGER`. The leading dimension of arrays *u*, *vt*, *difl*, *difr*, *poles*, *givnum*, and *z*. $ldu \geq n$.
ldgcol	`INTEGER`. The leading dimension of arrays *givcol* and *perm*. $ldgcol \geq n$.
work	`REAL` for `slasda` `DOUBLE PRECISION for dlasda`. Workspace array, `DIMENSION` $(6n + (smlsiz + 1)^2)$.

iwork INTEGER.
Workspace array, DIMENSION must be at least (7*n*).

Output Parameters

d On exit *d*, if *info* = 0, contains the singular values of the bidiagonal matrix.

u REAL for slasda
DOUBLE PRECISION for dlasda.
Array, DIMENSION (*ldu*, *smlsiz*) if *icompq* = 1.
Not referenced if *icompq* = 0.
If *icompq* = 1, on exit, *u* contains the left singular vector matrices of all subproblems at the bottom level.

vt REAL for slasda
DOUBLE PRECISION for dlasda.
Array, DIMENSION (*ldu*, *smlsiz*+1) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, *vt* contains the right singular vector matrices of all subproblems at the bottom level.

k INTEGER.
Array,
DIMENSION (*n*) if *icompq* = 1 and
DIMENSION (1) if *icompq* = 0.
If *icompq* = 1, on exit, *k*(*i*) is the dimension of the *i*-th secular equation on the computation tree.

difl REAL for slasda
DOUBLE PRECISION for dlasda.
Array, DIMENSION (*ldu*, *nlvl*),
where *nlvl* = floor ($\log_2$ (*n*/*smlsiz*))).

difr REAL for slasda
DOUBLE PRECISION for dlasda.
Array,
DIMENSION (*ldu*, 2 *nlvl*) if *icompq* = 1 and
DIMENSION (*n*) if *icompq* = 0.
If *icompq* = 1, on exit, *difl*(1:*n*, *i*) and *difr*(1:*n*,2*i* -1) record distances between singular values on the *i*-th level and singular values on the (*i* -1)-th level, and *difr*(1:*n*, 2*i*) contains the normalizing factors for the right singular vector matrix. See ?lasd8 for details.

z
REAL for `slasda`
DOUBLE PRECISION for `dlasda`.
Array,
DIMENSION (*ldu*, *nlvl*) if *icompq* = 1 and
DIMENSION (*n*) if *icompq* = 0.
The first *k* elements of *z*(1, *i*) contain the components of the deflation-adjusted updating row vector for subproblems on the *i*-th level.

poles
REAL for `slasda`
DOUBLE PRECISION for `dlasda`
Array, DIMENSION (*ldu*, 2**nlvl*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, *poles*(1, 2*i* - 1) and *poles*(1, 2*i*) contain the new and old singular values involved in the secular equations on the *i*-th level.

givptr
INTEGER.
Array, DIMENSION (*n*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, *givptr*(*i*) records the number of Givens rotations performed on the *i*-th problem on the computation tree.

givcol
INTEGER .
Array, DIMENSION (*ldgcol*, 2**nlvl*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, for each *i*, *givcol*(1, 2 *i* - 1) and *givcol*(1, 2 *i*) record the locations of Givens rotations performed on the *i*-th level on the computation tree.

perm
INTEGER .
Array, DIMENSION (*ldgcol*, *nlvl*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, *perm* (1, *i*) records permutations done on the *i*-th level of the computation tree.

givnum
REAL for `slasda`
DOUBLE PRECISION for `dlasda`.
Array DIMENSION (*ldu*, 2**nlvl*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, for each *i*, *givnum*(1, 2 *i* - 1) and *givnum*(1, 2 *i*) record the *C*- and *S*-values of Givens rotations performed on the *i*-th level on the computation tree.

c
REAL for `slasda`
DOUBLE PRECISION for `dlasda`.
Array,
DIMENSION (*n*) if *icompq* = 1, and
DIMENSION (1) if *icompq* = 0.

If *icompq* = 1 and the *i*-th subproblem is not square, on exit, *c*(*i*) contains the *C*-value of a Givens rotation related to the right null space of the *i*-th subproblem.

s — REAL for slasda
DOUBLE PRECISION for dlasda.
Array,
DIMENSION (*n*) *icompq* = 1, and
DIMENSION (1) if *icompq* = 0.
If *icompq* = 1 and the *i*-th subproblem is not square, on exit, *s*(*i*) contains the *S*-value of a Givens rotation related to the right null space of the *i*-th subproblem.

info — INTEGER.
= 0: successful exit.
< 0: if *info* = -*i*, the *i*-th argument had an illegal value > 0: if *info* = 1, an singular value did not converge

?lasdq

Computes the SVD of a real bidiagonal matrix with diagonal d *and off-diagonal* e*.*
Used by ?bdsdc.

Syntax

```
call slasdq ( uplo, sqre, n, ncvt, nru, ncc, d, e, vt,
              ldvt, u, ldu, c, ldc, work, info )
call dlasdq ( uplo, sqre, n, ncvt, nru, ncc, d, e, vt,
              ldvt, u, ldu, c, ldc, work, info )
```

Description

The routine ?lasdq computes the singular value decomposition (*SVD*) of a real (upper or lower) bidiagonal matrix with diagonal *d* and off-diagonal *e*, accumulating the transformations if desired. Letting *B* denote the input bidiagonal matrix, the algorithm computes orthogonal matrices *Q* and *P* such that $B = Q S P'$ (P' denotes the transpose of *P*). The singular values *S* are overwritten on *d*.
The input matrix *U* is changed to *UQ* if desired.
The input matrix *VT* is changed to $P' VT$ if desired.
The input matrix *C* is changed to $Q' C$ if desired.

Input Parameters

uplo — `CHARACTER*1`. On entry, *uplo* specifies whether the input bidiagonal matrix is upper or lower bidiagonal.
If *uplo* = 'U' or 'u' , *B* is upper bidiagonal;
If *uplo* = 'L' or 'l' , *B* is lower bidiagonal.

sqre — `INTEGER`.
= 0: then the input matrix is *n*-by-*n*.
= 1: then the input matrix is *n*-by-(*n*+1) if *uplu* = 'U' and (*n*+1)-by-*n* if *uplu* = 'L'. The bidiagonal matrix has *n* = *nl* + *nr* + 1 rows and *m* = *n* + *sqre* ≥ *n* columns.

n — `INTEGER`. On entry, *n* specifies the number of rows and columns in the matrix. *n* must be at least 0.

ncvt — `INTEGER`. On entry, *ncvt* specifies the number of columns of the matrix *VT*. *ncvt* must be at least 0.

nru — `INTEGER`. On entry, *nru* specifies the number of rows of the matrix *U*. *nru* must be at least 0.

ncc — `INTEGER`. On entry, *ncc* specifies the number of columns of the matrix *C*. *ncc* must be at least 0.

d — `REAL` for `slasdq`
`DOUBLE PRECISION` for `dlasdq`.
Array, `DIMENSION` (*n*). On entry, *d* contains the diagonal entries of the bidiagonal matrix whose *SVD* is desired.

e — `REAL` for `slasdq`
`DOUBLE PRECISION` for `dlasdq`.
Array, `DIMENSION` is (*n*-1) if *sqre* = 0 and *n* if *sqre* = 1. On entry, the entries of *e* contain the off-diagonal entries of the bidiagonal matrix whose *SVD* is desired.

vt — `REAL` for `slasdq`
`DOUBLE PRECISION` for `dlasdq`.
Array, `DIMENSION` (*ldvt*, *ncvt*). On entry, contains a matrix which on exit has been premultiplied by *P*', dimension *n*-by-*ncvt* if *sqre* = 0 and (*n*+1)-by-*ncvt* if *sqre* = 1 (not referenced if *ncvt*=0).

ldvt — `INTEGER`. On entry, *ldvt* specifies the leading dimension of *vt* as declared in the calling (sub) program. *ldvt* must be at least 1. If *ncvt* is nonzero, *ldvt* must also be at least *n*.

u — `REAL` for `slasdq`
`DOUBLE PRECISION` for `dlasdq`.
Array, `DIMENSION` (*ldu*, *n*). On entry, contains a matrix which on exit has been postmultiplied by *Q*, dimension *nru*-by-*n* if *sqre* = 0 and *nru*-by-(*n*+1) if *sqre* = 1 (not referenced if *nru*=0).

ldu — `INTEGER`.On entry, *ldu* specifies the leading dimension of *u* as declared in the calling (sub) program. *ldu* must be at least max(1, *nru*) .

c — `REAL` for `slasdq`
`DOUBLE PRECISION` for `dlasdq`.
Array, `DIMENSION` (*ldc*, *ncc*). On entry, contains an *n*-by-*ncc* matrix which on exit has been premultiplied by *Q'*, dimension *n*-by-*ncc* if *sqre* = 0 and (*n*+1)-by-*ncc* if *sqre* = 1 (not referenced if *ncc*=0).

ldc — `INTEGER`. On entry, *ldc* specifies the leading dimension of *c* as declared in the calling (sub) program. *ldc* must be at least 1. If *ncc* is non-zero, *ldc* must also be at least *n*.

work — `REAL` for `slasdq`
`DOUBLE PRECISION` for `dlasdq`.
Array, `DIMENSION` (4*n*).This is a workspace array. Only referenced if one of *ncvt*, *nru*, or *ncc* is nonzero, and if *n* is at least 2.

Output Parameters

d — On normal exit, *d* contains the singular values in ascending order.

e — On normal exit, *e* will contain 0. If the algorithm does not converge, *d* and *e* will contain the diagonal and superdiagonal entries of a bidiagonal matrix orthogonally equivalent to the one given as input.

vt — On exit, the matrix has been premultiplied by *P*'.

u — On exit, the matrix has been postmultiplied by *Q*.

c — On exit, the matrix has been premultiplied by *Q*'.

info — `INTEGER`. On exit, a value of 0 indicates a successful exit. If *info* < 0, argument number -*info* is illegal. If *info* > 0, the algorithm did not converge, and *info* specifies how many superdiagonals did not converge.

?lasdt

Creates a tree of subproblems for bidiagonal divide and conquer.
Used by `?bdsdc`.

Syntax

```
call slasdt ( n, lvl, nd, inode, ndiml, ndimr, msub )
call dlasdt ( n, lvl, nd, inode, ndiml, ndimr, msub )
```

Description

The routine creates a tree of subproblems for bidiagonal divide and conquer.

Input Parameters

n — `INTEGER`. On entry, the number of diagonal elements of the bidiagonal matrix.

msub — `INTEGER`. On entry, the maximum row dimension each subproblem at the bottom of the tree can be of.

Output Parameters

lvl — `INTEGER`. On exit, the number of levels on the computation tree.

nd — `INTEGER`. On exit, the number of nodes on the tree.

inode — `INTEGER`.
Array, `DIMENSION` (*n*). On exit, centers of subproblems.

ndiml — `INTEGER` .
Array, `DIMENSION` (*n*). On exit, row dimensions of left children.

ndimr — `INTEGER` .
Array, `DIMENSION` (*n*). On exit, row dimensions of right children.

?laset

Initializes the off-diagonal elements and the diagonal elements of a matrix to given values.

Syntax

```
call slaset ( uplo, m, n, alpha, beta, a, lda )
call dlaset ( uplo, m, n, alpha, beta, a, lda )
call claset ( uplo, m, n, alpha, beta, a, lda )
call zlaset ( uplo, m, n, alpha, beta, a, lda )
```

Description

The routine initializes an *m*-by-*n* matrix *A* to *beta* on the diagonal and *alpha* on the off-diagonals .

Input parameters

uplo — `CHARACTER*1`. Specifies the part of the matrix *A* to be set.
If *uplo* = 'U', upper triangular part is set; the strictly lower triangular part of *A* is not changed.
If *uplo* = 'L': lower triangular part is set; the strictly upper triangular part of *A* is not changed.
Otherwise: all of the matrix *A* is set.

m — `INTEGER`. The number of rows of the matrix *A*. $m \geq 0$.

n — `INTEGER`. The number of columns of the matrix *A*.
$n \geq 0$.

alpha, beta — `REAL` for `slaset`
`DOUBLE PRECISION` for `dlaset`
`COMPLEX` for `claset`
`COMPLEX*16` for `zlaset`.
The constants to which the off-diagonal and diagonal elements are to be set, respectively.

a — `REAL` for `slaset`
`DOUBLE PRECISION` for `dlaset`
`COMPLEX` for `claset`

COMPLEX*16 for zlaset.
Array, DIMENSION (*lda*, *n*).
On entry, the *m*-by-*n* matrix *A*.

lda INTEGER. The leading dimension of the array *A*.
lda ≥ max(1,*m*).

Output Parameters

a On exit, the leading *m*-by-*n* submatrix of *A* is set as follows:
if *uplo* = 'U', $A(i,j)$ = *alpha*, $1\leq i\leq j-1$, $1\leq j\leq n$,
if *uplo* = 'L', $A(i,j)$ = *alpha*, $j+1\leq i\leq m$, $1\leq j\leq n$,
otherwise, $A(i,j)$ = *alpha*, $1\leq i\leq m$, $1\leq j\leq n$, $i\neq j$,

and, for all *uplo*, $A(i,i)$ = *beta*, $1\leq i\leq \min(m, n)$.

?lasq1

Computes the singular values of a real square bidiagonal matrix. Used by ?bdsqr.

Syntax

```
call slasq1 ( n, d, e, work, info )
call dlasq1 ( n, d, e, work, info )
```

Description

The routine ?lasq1 computes the singular values of a real *n*-by-*n* bidiagonal matrix with diagonal *d* and off-diagonal *e*. The singular values are computed to high relative accuracy, in the absence of denormalization, underflow and overflow.

Input Parameters

n INTEGER.The number of rows and columns in the matrix. *n* ≥ 0.

d REAL for slasq1
DOUBLE PRECISION for dlasq1.
Array, DIMENSION (*n*). On entry, *d* contains the diagonal elements of the bidiagonal matrix whose *SVD* is desired.

e REAL for slasq1
DOUBLE PRECISION for dlasq1.
Array, DIMENSION (*n*). On entry, elements *e*(1:*n*-1) contain the off-diagonal elements of the bidiagonal matrix whose *SVD* is desired.

work REAL for slasq1
DOUBLE PRECISION for dlasq1.
Workspace array, DIMENSION (4*n*).

Output Parameters

d On normal exit, *d* contains the singular values in decreasing order.

e On exit, *e* is overwritten.

info INTEGER.
= 0: successful exit;
< 0: if *info* = -*i*, the *i*-th argument had an illegal value; > 0: the algorithm failed:
= 1, a split was marked by a positive value in *e*;
= 2, current block of *z* not diagonalized after 30**n* iterations (in inner while loop);
= 3, termination criterion of outer while loop not met (program created more than *n* unreduced blocks.

?lasq2

Computes all the eigenvalues of the symmetric positive definite tridiagonal matrix associated with the qd array z *to high relative accuracy. Used by* ?bdsqr *and* ?stegr.

Syntax

```
call slasq2 ( n, z, info )
call dlasq2 ( n, z, info )
```

Description

The routine ?lasq2 computes all the eigenvalues of the symmetric positive definite tridiagonal matrix associated with the *qd* array *z* to high relative accuracy, in the absence of denormalization, underflow and overflow.

To see the relation of *z* to the tridiagonal matrix, let *L* be a unit lower bidiagonal matrix with subdiagonals *z*(2,4,6,,..) and let *U* be an upper bidiagonal matrix with 1's above and diagonal *z*(1,3,5,,..). The tridiagonal is *LU* or, if you prefer, the symmetric tridiagonal to which it is similar.

Input Parameters

n INTEGER. The number of rows and columns in the matrix. $n \geq 0$.

z REAL for slasq2
DOUBLE PRECISION for dlasq2.
Array, DIMENSION (4*n*). On entry, *z* holds the *qd* array.

Output Parameters

z On exit, entries 1 to *n* hold the eigenvalues in decreasing order, *z*(2*n*+1) holds the trace, and *z*(2*n*+2) holds the sum of the eigenvalues. If $n > 2$, then *z*(2*n*+3) holds the iteration count, *z*(2*n*+4) holds ndivs/nin^2, and *z*(2*n*+5) holds the percentage of shifts that failed.

info INTEGER.
= 0: successful exit;
< 0: if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*, if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*);
> 0: the algorithm failed:
= 1, a split was marked by a positive value in *e*;
= 2, current block of *z* not diagonalized after 30**n* iterations (in inner while loop);
= 3, termination criterion of outer while loop not met (program created more than *n* unreduced blocks).

Application Notes

The routine ?lasq2 defines a logical variable, *ieee*, which is .TRUE. on machines which follow IEEE-754 floating-point standard in their handling of infinities and NaNs, and .FALSE. otherwise. This variable is passed to ?lasq3.

?lasq3

Checks for deflation, computes a shift and calls dqds. Used by `?bdsqr`.

Syntax

```
call slasq3 ( i0, n0, z, pp, dmin, sigma, desig, qmax, nfail, iter, ndiv,
    ieee )
call dlasq3 ( i0, n0, z, pp, dmin, sigma, desig, qmax, nfail, iter, ndiv,
    ieee )
```

Description

The routine `?lasq3` checks for deflation, computes a shift (*tau*) and calls *dqds*.In case of failure, it changes shifts, and tries again until output is positive.

Input Parameters

i0	`INTEGER`. First index.
n0	`INTEGER`. Last index.
z	`REAL` for `slasq3` `DOUBLE PRECISION` for `dlasq3`. Array, `DIMENSION` (4*n*). *z* holds the *qd* array.
pp	`INTEGER`. *pp*=0 for ping, *pp*=1 for pong.
desig	`REAL` for `slasq3` `DOUBLE PRECISION` for `dlasq3`. Lower order part of *sigma*.
qmax	`REAL` for `slasq3` `DOUBLE PRECISION` for `dlasq3`. Maximum value of *q*.
ieee	`LOGICAL`. Flag for IEEE or non-IEEE arithmetic (passed to `?lasq5`).

Output Parameters

dmin	`REAL` for `slasq3` `DOUBLE PRECISION` for `dlasq3`. Minimum value of *d*.
sigma	`REAL` for `slasq3` `DOUBLE PRECISION` for `dlasq3`. Sum of shifts used in current segment.
desig	Lower order part of *sigma*.
nfail	`INTEGER`. Number of times shift was too big.
iter	`INTEGER`. Number of iterations.
ndiv	`INTEGER`. Number of divisions.
ttype	`INTEGER`. Shift type.

?lasq4

Computes an approximation to the smallest eigenvalue using values of d from the previous transform. Used by `?bdsqr`.

Syntax

```
call slasq4 ( i0, n0, z, pp, n0in, dmin, dmin1, dmin2, dn, dn1, dn2, tau,
    ttype )
call dlasq4 ( i0, n0, z, pp, n0in, dmin, dmin1, dmin2, dn, dn1, dn2, tau,
    ttype )
```

Description

The routine computes an approximation *tau* to the smallest eigenvalue using values of *d* from the previous transform.

Input Parameters

i0	`INTEGER`. First index.
n0	`INTEGER`. Last index.

z	REAL for slasq4 DOUBLE PRECISION for dlasq4. Array, DIMENSION (4*n*). *z* holds the *qd* array.
pp	INTEGER. *pp*=0 for ping, *pp*=1 for pong.
noin	INTEGER. The value of *n0* at start of eigtest.
dmin	REAL for slasq4 DOUBLE PRECISION for dlasq4. Minimum value of *d*.
dmin1	REAL for slasq4 DOUBLE PRECISION for dlasq4. Minimum value of *d*, excluding *d*(*n0*).
dmin2	REAL for slasq4 DOUBLE PRECISION for dlasq4. Minimum value of *d*, excluding *d*(*n0*) and *d*(*n0*-1).
dn	REAL for slasq4 DOUBLE PRECISION for dlasq4. Contains *d*(*n*).
dn1	REAL for slasq4 DOUBLE PRECISION for dlasq4. Contains *d*(*n*-1).
dn2	REAL for slasq4 DOUBLE PRECISION for dlasq4. Contains *d*(*n*-2).

Output Parameters

tau	REAL for slasq4 DOUBLE PRECISION for dlasq4. This is the shift.
ttype	INTEGER. Shift type.

?lasq5

Computes one dqds transform in ping-pong form. Used by `?bdsqr` *and* `?stegr`.

Syntax

```
call slasq5 ( i0, n0, z, pp, tau, dmin, dmin1, dmin2, dn, dnm1, dnm2,
    ieee )
call dlasq5 ( i0, n0, z, pp, tau, dmin, dmin1, dmin2, dn, dnm1, dnm2,
    ieee )
```

Description

The routine computes one *dqds* transform in ping-pong form, one version for IEEE machines another for non-IEEE machines.

Input Parameters

i0	`INTEGER` First index.
n0	`INTEGER` Last index.
z	`REAL` for `slasq5` `DOUBLE PRECISION` for `dlasq5`. Array, `DIMENSION` (4*n*). *z* holds the *qd* array. *emin* is stored in *z*(4**n0*) to avoid an extra argument.
pp	`INTEGER`. *pp*=0 for ping, *pp*=1 for pong.
tau	`REAL` for `slasq5` `DOUBLE PRECISION` for `dlasq5`. This is the shift.
ieee	`LOGICAL.` Flag for IEEE or non-IEEE arithmetic.

Output Parameters

dmin	`REAL` for `slasq5` `DOUBLE PRECISION` for `dlasq5`. Minimum value of *d*.

dmin1 — `REAL` for `slasq5`
`DOUBLE PRECISION` for `dlasq5`.
Minimum value of *d*, excluding *d*(*n0*).

dmin2 — `REAL` for `slasq5`
`DOUBLE PRECISION` for `dlasq5`.
Minimum value of *d*, excluding *d*(*n0*) and *d*(*n0*-1).

dn — `REAL` for `slasq5`
`DOUBLE PRECISION` for `dlasq5`.
Contains *d*(*n0*), the last value of *d*.

dnm1 — `REAL` for `slasq5`
`DOUBLE PRECISION` for `dlasq5`.
Contains *d*(*n0*-1).

dnm2 — `REAL` for `slasq5`
`DOUBLE PRECISION` for `dlasq5`.
Contains *d*(*n0*-2).

?lasq6

Computes one dqds transform in ping-pong form. Used by `?bdsqr` *and* `?stegr`.

Syntax

```
call slasq6 ( i0, n0, z, pp, dmin, dmin1, dmin2, dn, dnm1, dnm2 )
call dlasq6 ( i0, n0, z, pp, dmin, dmin1, dmin2, dn, dnm1, dnm2 )
```

Description

The routine `?lasq6` computes one *dqd* (shift equal to zero) transform in ping-pong form, with protection against underflow and overflow.

Input Parameters

i0 — `INTEGER`. First index.

n0 — `INTEGER`. Last index.

z	`REAL` for `slasq6` `DOUBLE PRECISION` for `dlasq6`. Array, `DIMENSION` (4*n*). *z* holds the *qd* array. *emin* is stored in *z*(4**n0*) to avoid an extra argument.
pp	`INTEGER`. *pp*=0 for ping, *pp*=1 for pong.

Output Parameters

dmin	`REAL` for `slasq6` `DOUBLE PRECISION` for `dlasq6`. Minimum value of *d*.
dmin1	`REAL` for `slasq6` `DOUBLE PRECISION` for `dlasq6`. Minimum value of *d*, excluding *d*(*n0*).
dmin2	`REAL` for `slasq6` `DOUBLE PRECISION` for `dlasq6`. Minimum value of *d*, excluding *d*(*n0*) and *d*(*n0*-1).
dn	`REAL` for `slasq6` `DOUBLE PRECISION` for `dlasq6`. Contains *d*(*n0*), the last value of *d*.
dnm1	`REAL` for `slasq6` `DOUBLE PRECISION` for `dlasq6`. Contains *d*(*n0*-1).
dnm2	`REAL` for `slasq6` `DOUBLE PRECISION` for `dlasq6`. Contains *d*(*n0*-2).

?lasr

Applies a sequence of plane rotations to a general rectangular matrix.

Syntax

```
call slasr ( side, pivot, direct, m, n, c, s, a, lda )
call dlasr ( side, pivot, direct, m, n, c, s, a, lda )
```

```
call clasr ( side, pivot, direct, m, n, c, s, a, lda )
call zlasr ( side, pivot, direct, m, n, c, s, a, lda )
```

Description

The routine performs the transformation:

$A := P A$, when `side` = 'L' or 'l' (Left-hand side)

$A := A P'$, when `side` = 'R' or 'r' (Right-hand side)

where A is an `m`-by-`n` real matrix and P is an orthogonal matrix, consisting of a sequence of plane rotations determined by the parameters `pivot` and `direct` as follows (z = `m` when `side` = 'L' or 'l' and z = `n` when `side` = 'R' or 'r'):

When `direct` = 'F' or 'f' (Forward sequence) then

$P = P(z-1) \dots P(2) P(1)$,

and when `direct` = 'B' or 'b' (Backward sequence) then

$P = P(1) P(2) \dots P(z-1)$,

where $P(k)$ is a plane rotation matrix for the following planes:

when `pivot` = 'V' or 'v' (Variable pivot), the plane (k, $k+1$)
when `pivot` = 'T' or 't' (Top pivot), the plane (1, $k+1$)
when `pivot` = 'B' or 'b' (Bottom pivot), the plane (k, z)

`c`(k) and `s`(k) must contain the cosine and sine that define the matrix $P(k)$. The 2-by-2 plane rotation part of the matrix $P(k)$, $R(k)$, is assumed to be of the form:

$$R(k) = \begin{bmatrix} c(k) & s(k) \\ -s(k) & c(k) \end{bmatrix}$$

Input Parameters

side — `CHARACTER*1`. Specifies whether the plane rotation matrix P is applied to A on the left or the right.
= 'L': Left, compute $A := P A$
= 'R': Right, compute $A := A P'$

direct — `CHARACTER*1`. Specifies whether P is a forward or backward sequence of plane rotations.
= 'F': Forward, $P = P(z-1) \dots P(2) P(1)$
= 'B': Backward, $P = P(1) P(2) \dots P(z-1)$

pivot	`CHARACTER*1`. Specifies the plane for which $P(k)$ is a plane rotation matrix. = 'V': Variable pivot, the plane $(k, k+1)$ = 'T': Top pivot, the plane $(1, k+1)$ = 'B': Bottom pivot, the plane (k, z)
m	`INTEGER`. The number of rows of the matrix A. If $m \le 1$, an immediate return is effected.
n	`INTEGER`. The number of columns of the matrix A. If $n \le 1$, an immediate return is effected.
c, s	`REAL` for `slasr/clasr` `DOUBLE PRECISION` for `dlasr/zlasr`. Arrays, `DIMENSION` $(m-1)$ if *side* = 'L', $(n-1)$ if *side* = 'R'. $c(k)$ and $s(k)$ contain the cosine and sine that define the matrix $P(k)$ as described above.
a	`REAL` for `slasr` `DOUBLE PRECISION` for `dlasr` `COMPLEX` for `clasr` `COMPLEX*16` for `zlasr`. Array, `DIMENSION` (*lda*, *n*). The *m*-by-*n* matrix A.
lda	`INTEGER`. The leading dimension of the array A. $lda \ge \max(1,m)$.

Output Parameters

a	On exit, A is overwritten by PA if *side* = 'R' or by AP' if *side* = 'L'.

?lasrt

Sorts numbers in increasing or decreasing order.

Syntax

```
call slasrt ( id, n, d, info )
call dlasrt ( id, n, d, info )
```

Description

The routine `?lasrt` sorts the numbers in *d* in increasing order (if *id* = 'I') or in decreasing order (if *id* = 'D'). It uses Quick Sort, reverting to Insertion Sort on arrays of size ≤ 20. Dimension of `stack` limits *n* to about 2^{32}.

Input Parameters

id — `CHARACTER*1`.
= 'I': sort *d* in increasing order;
= 'D': sort *d* in decreasing order.

n — `INTEGER`. The length of the array *d*.

d — `REAL` for `slasrt`
`DOUBLE PRECISION` for `dlasrt`.
On entry, the array to be sorted.

Output Parameters

d — On exit, *d* has been sorted into increasing order ($d(1) \le \ldots \le d(n)$) or into decreasing order ($d(1) \ge \ldots \ge d(n)$), depending on *id*.

info — `INTEGER`.
= 0: successful exit
< 0: if *info* = -i, the *i*-th argument had an illegal value.

?lassq

Updates a sum of squares represented in scaled form.

Syntax

```
call slassq ( n, x, incx, scale, sumsq )
call dlassq ( n, x, incx, scale, sumsq )
call classq ( n, x, incx, scale, sumsq )
call zlassq ( n, x, incx, scale, sumsq )
```

Description

The real routines `slassq/dlassq` return the values *scl* and *smsq* such that

$$scl^2 * smsq = x(1)^2 + \ldots + x(n)^2 + scale^2 * sumsq,$$

where $x(i) = x(1 + (i - 1)\ incx)$.
The value of *sumsq* is assumed to be non-negative and *scl* returns the value
$scl = \max(scale, \mathrm{abs}(x(i)))$.

Values *scale* and *sumsq* must be supplied in `scale` and `sumsq`, and *scl* and *smsq* are overwritten on `scale` and `sumsq`, respectively.

The complex routines `classq/zlassq` return the values *scl* and *ssq* such that

$$scl^2 * ssq = x(1)^2 + \ldots + x(n)^2 + scale^2 * sumsq,$$

where $x(i) = \mathrm{abs}(x(1 + (i - 1)\ incx))$.
The value of *sumsq* is assumed to be at least unity and the value of *ssq* will then satisfy
$1.0 \le ssq \le sumsq + 2n$

scale is assumed to be non-negative and *scl* returns the value

$$scl = \max_i(scale, \mathrm{abs}(\mathrm{real}(x(i))), \quad \mathrm{abs}(\mathrm{aimag}(x(i)))).$$

Values *scale* and *sumsq* must be supplied in `scale` and `sumsq`, and *scl* and *ssq* are overwritten on `scale` and `sumsq`, respectively.

All routines `?lassq` make only one pass through the vector *x*.

Input Parameters

n — `INTEGER`. The number of elements to be used from the vector *x*.

x — `REAL` for `slassq`
`DOUBLE PRECISION` for `dlassq`
`COMPLEX` for `classq`
`COMPLEX*16` for `zlassq`.
The vector for which a scaled sum of squares is computed: $x(i) = x(1 + (i - 1)\ incx)$, $1 \le i \le n$.

incx — `INTEGER`. The increment between successive values of the vector *x*. $incx > 0$.

scale — `REAL` for `slassq/classq`
`DOUBLE PRECISION` for `dlassq/zlassq`.
On entry, the value *scale* in the equation above.

sumsq — `REAL` for `slassq/classq`
`DOUBLE PRECISION` for `dlassq/zlassq`.
On entry, the value *sumsq* in the equation above.

Output Parameters

scale — On exit, *scale* is overwritten with *scl*, the scaling factor for the sum of squares.

sumsq — *For real flavors:*
On exit, *sumsq* is overwritten with the value *smsq* in the equation above.
For complex flavors:
On exit, *sumsq* is overwritten with the value *ssq* in the equation above.

?lasv2

Computes the singular value decomposition of a 2-by-2 triangular matrix.

Syntax

```
call slasv2 ( f, g, h, ssmin, ssmax, snr, csr, snl, csl )
call dlasv2 ( f, g, h, ssmin, ssmax, snr, csr, snl, csl )
```

Description

The routine `?lasv2` computes the singular value decomposition of a 2-by-2 triangular matrix

$$\begin{bmatrix} f & g \\ 0 & h \end{bmatrix}$$

On return, abs(*ssmax*) is the larger singular value, abs(*ssmin*) is the smaller singular value, and (*csl*,*snl*) and (*csr*,*snr*) are the left and right singular vectors for abs(*ssmax*), giving the decomposition

$$\begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} f & g \\ 0 & h \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix} = \begin{bmatrix} ssmax & 0 \\ 0 & ssmin \end{bmatrix}$$

Input Parameters

f, g, h — `REAL` for `slasv2`
`DOUBLE PRECISION` for `dlasv2`.
The (1,1), (1,2) and (2,2) elements of the 2-by-2 matrix, respectively.

Output Parameters

ssmin, ssmax — `REAL` for `slasv2`
`DOUBLE PRECISION` for `dlasv2`.
abs(*ssmin*) and abs(*ssmax*) is the smaller and the larger singular value, respectively.

snl, csl — `REAL` for `slasv2`
`DOUBLE PRECISION` for `dlasv2`.
The vector (*csl*, *snl*) is a unit left singular vector for the singular value abs(*ssmax*).

snr, csr — `REAL` for `slasv2`
`DOUBLE PRECISION` for `dlasv2`.
The vector (*csr*, *snr*) is a unit right singular vector for the singular value abs(*ssmax*).

Application Notes

Any input parameter may be aliased with any output parameter.
Barring over/underflow and assuming a guard digit in subtraction, all output quantities are correct to within a few units in the last place (ulps).

In IEEE arithmetic, the code works correctly if one matrix element is infinite.
Overflow will not occur unless the largest singular value itself overflows or is within a few ulps of overflow. (On machines with partial overflow, like the Cray, overflow may occur if the largest singular value is within a factor of 2 of overflow.)
Underflow is harmless if underflow is gradual. Otherwise, results may correspond to a matrix modified by perturbations of size near the underflow threshold.

?laswp

Performs a series of row interchanges on a general rectangular matrix.

Syntax

```
call slaswp ( n, a, lda, k1, k2, ipiv, incx )
call dlaswp ( n, a, lda, k1, k2, ipiv, incx )
call claswp ( n, a, lda, k1, k2, ipiv, incx )
call zlaswp ( n, a, lda, k1, k2, ipiv, incx )
```

Description

The routine performs a series of row interchanges on the matrix *A*. One row interchange is initiated for each of rows *k1* through *k2* of *A*.

Input Parameters

n — INTEGER.The number of columns of the matrix *A*.

a — REAL for slaswp
DOUBLE PRECISION for dlaswp
COMPLEX for claswp
COMPLEX*16 for zlaswp.
Array, DIMENSION (*lda*, *n*).
On entry, the matrix of column dimension *n* to which the row interchanges will be applied.

lda — INTEGER. The leading dimension of the array *a*.

k1 — INTEGER. The first element of *ipiv* for which a row interchange will be done.

k2 — INTEGER. The last element of *ipiv* for which a row interchange will be done.

ipiv — INTEGER.
Array, DIMENSION (*m* * abs(*incx*)).
The vector of pivot indices. Only the elements in positions *k1* through *k2* of *ipiv* are accessed.
ipiv(*k*) = *l* implies rows *k* and *l* are to be interchanged.

incx — INTEGER. The increment between successive values of *ipiv*. If *ipiv* is negative, the pivots are applied in reverse order.

Output Parameters

a	On exit, the permuted matrix.

?lasy2

Solves the Sylvester matrix equation where the matrices are of order 1 or 2.

Syntax

```
call slasy2 ( ltranl, ltranr, isgn, n1, n2, tl, ldtl, tr,ldtr, b, ldb,
    scale, x, ldx, xnorm, info )
call dlasy2 ( ltranl, ltranr, isgn, n1, n2, tl, ldtl, tr,ldtr, b, ldb,
    scale, x, ldx, xnorm, info )
```

Description

The routine solves for the `n1`-by-`n2` matrix X, $1 \le$ `n1`, `n2` ≤ 2, in

$$op(TL) * X + isgn * X * op(TR) = scale * B,$$

where
TL is `n1`-by-`n1`,
TR is `n2`- by-`n2`,
B is `n1`-by-`n2`,
and *isgn* = 1 or -1. op(T) = T or T', where T' denotes the transpose of T.

Input Parameters

ltranl	`LOGICAL`. On entry, *ltranl* specifies the op(TL): = `.FALSE.`, op(TL) = TL, = `.TRUE.`, op(TL) = TL'.
ltranr	`LOGICAL`. On entry, *ltranr* specifies the op(TR): = `.FALSE.`, op(TR) = TR, = `.TRUE.`, op(TR) = TR'.
isgn	`INTEGER`. On entry, *isgn* specifies the sign of the equation as described before. *isgn* may only be 1 or -1.

n1	INTEGER. On entry, *n1* specifies the order of matrix *TL*. *n1* may only be 0, 1 or 2.
n2	INTEGER. On entry, *n2* specifies the order of matrix *TR*. *n2* may only be 0, 1 or 2.
tl	REAL for slasy2 DOUBLE PRECISION for dlasy2. Array, DIMENSION (*ldtl*,2). On entry, *tl* contains an *n1*-by-*n1* matrix *TL*.
ldtl	INTEGER.The leading dimension of the matrix *tl*. *ldtl* ≥ max(1,*n1*).
tr	REAL for slasy2 DOUBLE PRECISION for dlasy2. Array, DIMENSION (*ldtr*,2). On entry, *tr* contains an *n2*-by-*n2* matrix *TR*.
ldtr	INTEGER. The leading dimension of the matrix *tr*. *ldtr* ≥ max(1,*n2*).
b	REAL for slasy2 DOUBLE PRECISION for dlasy2. Array, DIMENSION (*ldb*,2). On entry, the *n1*-by-*n2* matrix *b* contains the right-hand side of the equation.
ldb	INTEGER. The leading dimension of the matrix *b*. *ldb* ≥ max(1,*n1*).
ldx	INTEGER. The leading dimension of the output matrix *x*. *ldx* ≥ max(1,*n1*).

Output Parameters

scale	REAL for slasy2 DOUBLE PRECISION for dlasy2. On exit, *scale* contains the scale factor. *scale* is chosen less than or equal to 1 to prevent the solution overflowing.
x	REAL for slasy2 DOUBLE PRECISION for dlasy2. Array, DIMENSION (*ldx*,2). On exit, *x* contains the *n1*-by-*n2* solution.

xnorm — `REAL` for `slasy2`
`DOUBLE PRECISION` for `dlasy2`.
On exit, *xnorm* is the infinity-norm of the solution.

info — `INTEGER`. On exit, *info* is set to
0: successful exit.
1: *TL* and *TR* have too close eigenvalues, so *TL* or *TR* is perturbed to get a nonsingular equation.

NOTE. In the interests of speed, this routine does not check the inputs for errors.

?lasyf

Computes a partial factorization of a real/complex symmetric matrix, using the diagonal pivoting method.

Syntax

```
call slasyf ( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info)
call dlasyf ( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info)
call clasyf ( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info)
call zlasyf ( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info)
```

Description

The routine `?lasyf` computes a partial factorization of a real/complex symmetric matrix *A* using the Bunch-Kaufman diagonal pivoting method. The partial factorization has the form:

$$A = \begin{bmatrix} I & U_{12} \\ 0 & U_{22} \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ U_{12}' & U_{22}' \end{bmatrix} \quad \text{if } uplo = \text{'U', or}$$

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} L_{11}' & L_{21}' \\ 0 & I \end{bmatrix} \quad \text{if } uplo = 'L'$$

where the order of *D* is at most *nb*. The actual order is returned in the argument *kb*, and is either *nb* or *nb*-1, or *n* if $n \le nb$.

This is an auxiliary routine called by `?sytrf`. It uses blocked code (calling Level 3 BLAS) to update the submatrix A_{11} (if *uplo* = 'U') or A_{22} (if *uplo* = 'L').

Input Parameters

uplo `CHARACTER*1`.
Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored:
= 'U': Upper triangular
= 'L': Lower triangular

n `INTEGER`.
The order of the matrix *A*. $n \ge 0$.

nb `INTEGER`.
The maximum number of columns of the matrix *A* that should be factored. *nb* should be at least 2 to allow for 2-by-2 pivot blocks.

a `REAL` for `slasyf`
`DOUBLE PRECISION` for `dlasyf`
`COMPLEX` for `clasyf`
`COMPLEX*16` for `zlasyf`.
Array, `DIMENSION` (*lda*, *n*). On entry, the symmetric matrix *A*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda `INTEGER`.
The leading dimension of the array *a*. $lda \ge \max(1,n)$.

w REAL for slasyf
DOUBLE PRECISION for dlasyf
COMPLEX for clasyf
COMPLEX*16 for zlasyf.
Workspace array, DIMENSION (*ldw*, *nb*).

ldw INTEGER.
The leading dimension of the array *w*. *ldw* ≥ max(1,*n*).

Output Parameters

kb INTEGER.
The number of columns of *A* that were actually factored *kb* is either *nb*-1 or *nb*, or *n* if *n* ≤ *nb*.

a On exit, *a* contains details of the partial factorization.

ipiv INTEGER.
Array, DIMENSION (*n*). Details of the interchanges and the block structure of *D*.
If *uplo* = 'U', only the last *kb* elements of *ipiv* are set;
if *uplo* = 'L', only the first *kb* elements are set.

If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) were interchanged and *D(k,k)* is a 1-by-1 diagonal block. If *uplo* = 'U' and *ipiv*(*k*) = *ipiv*(*k*-1) < 0, then rows and columns *k*-1and -*ipiv*(*k*) were interchanged and *D*(*k*-1:*k*, *k*-1:*k*) is a 2-by-2 diagonal block.
If *uplo* = 'L' and *ipiv*(*k*) = *ipiv*(*k*+1) < 0, then rows and columns *k*+1 and -*ipiv*(*k*) were interchanged and *D(k*:*k*+1, *k*:*k*+1) is a 2-by-2 diagonal block.

info INTEGER.
= 0: successful exit
> 0: if *info* = *k*, *D(k,k)* is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular.

?lahef

Computes a partial factorization of a complex Hermitian indefinite matrix, using the diagonal pivoting method.

Syntax

```
call clahef ( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info)
call zlahef ( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info)
```

Description

The routine `?lahef` computes a partial factorization of a complex Hermitian matrix *A*, using the Bunch-Kaufman diagonal pivoting method. The partial factorization has the form:

$$A = \begin{bmatrix} I & U_{12} \\ 0 & U_{22} \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ U_{12}' & U_{22}' \end{bmatrix}$$ if *uplo* = 'U', or

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} L_{11}' & L_{21}' \\ 0 & I \end{bmatrix}$$ if *uplo* = 'L'

where the order of *D* is at most *nb*. The actual order is returned in the argument *kb*, and is either *nb* or *nb*-1, or *n* if $n \le nb$.
Note that U' denotes the conjugate transpose of *U*.

This is an auxiliary routine called by `?hetrf`. It uses blocked code (calling Level 3 BLAS) to update the submatrix A_{11} (if *uplo* = 'U') or A_{22} (if *uplo* = 'L').

Input Parameters

uplo — `CHARACTER*1`.
Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is stored:
= 'U': Upper triangular
= 'L': Lower triangular

n — INTEGER.
The order of the matrix *A*. *n* ≥ 0.

nb — INTEGER.
The maximum number of columns of the matrix *A* that should be factored. *nb* should be at least 2 to allow for 2-by-2 pivot blocks.

a — COMPLEX for clahef
COMPLEX*16 for zlahef.
Array, DIMENSION (*lda*, *n*).
On entry, the Hermitian matrix *A*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda — INTEGER.
The leading dimension of the array *a*. *lda* ≥ max(1,*n*).

w — COMPLEX for clahef
COMPLEX*16 for zlahef.
Workspace array, DIMENSION (*ldw*, *nb*).

ldw — INTEGER.
The leading dimension of the array *w*. *ldw* ≥ max(1,*n*).

Output Parameters

kb — INTEGER.
The number of columns of *A* that were actually factored *kb* is either *nb*-1 or *nb*, or *n* if *n* ≤ *nb*.

a — On exit, *a* contains details of the partial factorization.

ipiv — INTEGER.
Array, DIMENSION (*n*). Details of the interchanges and the block structure of *D*.
If *uplo* = 'U', only the last *kb* elements of *ipiv* are set;
if *uplo* = 'L', only the first *kb* elements are set.

If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) were interchanged and *D(k,k)* is a 1-by-1 diagonal block. If *uplo* = 'U' and *ipiv*(*k*) = *ipiv*(*k*-1) < 0, then rows and columns *k*-1and -*ipiv*(*k*) were interchanged and *D*(*k*-1:*k*, *k*-1:*k*)

is a 2-by-2 diagonal block.
If `uplo` = 'L' and `ipiv`(*k*) = `ipiv`(*k*+1) < 0, then rows and columns *k*+1 and -`ipiv`(*k*) were interchanged and *D*(*k*:*k*+1, *k*:*k*+1) is a 2-by-2 diagonal block.

info INTEGER.
= 0: successful exit
> 0: if *info* = *k*, *D(k,k)* is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular.

?latbs

Solves a triangular banded system of equations.

Syntax

```
call slatbs ( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale,
    cnorm, info )
call dlatbs ( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale,
    cnorm, info )
call clatbs ( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale,
    cnorm, info )
call zlatbs ( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale,
    cnorm, info )
```

Description

The routine solves one of the triangular systems

$Ax = s\,b$ or $A^T x = s\,b$ or $A^H x = s\,b$ (for complex flavors)

with scaling to prevent overflow, where A is an upper or lower triangular band matrix. Here A^T denotes the transpose of A, A^H denotes the conjugate transpose of A, x and b are n-element vectors, and s is a scaling factor, usually less than or equal to 1, chosen so that the components of x will be less than the overflow threshold. If the unscaled problem will not cause overflow, the Level 2 BLAS routine `?tbsv` is called. If the matrix A is singular ($A(j, j) = 0$ for some j), then s is set to 0 and a non-trivial solution to $Ax = 0$ is returned.

Input Parameters

uplo CHARACTER*1.
Specifies whether the matrix A is upper or lower triangular.
= 'U': Upper triangular
= 'L': Lower triangular

trans CHARACTER*1.
Specifies the operation applied to A.
= 'N': Solve $Ax = s\, b$ (no transpose)
= 'T': Solve $A^T x = s\, b$ (transpose)
= 'C': Solve $A^H x = s\, b$ (conjugate transpose)

diag CHARACTER*1.
Specifies whether or not the matrix A is unit triangular
= 'N': Non-unit triangular
= 'U': Unit triangular

normin CHARACTER*1.
Specifies whether *cnorm* has been set or not.
= 'Y': *cnorm* contains the column norms on entry;
= 'N': *cnorm* is not set on entry. On exit, the norms will be computed and stored in *cnorm*.

n INTEGER.
The order of the matrix A. $n \geq 0$.

kd INTEGER.
The number of subdiagonals or superdiagonals in the triangular matrix A. $kd \geq 0$.

ab REAL for slatbs
DOUBLE PRECISION for dlatbs
COMPLEX for clatbs
COMPLEX*16 for zlatbs.
Array, DIMENSION (*ldab*, *n*). The upper or lower triangular band matrix A, stored in the first kd+1 rows of the array. The j-th column of A is stored in the j-th column of the array *ab* as follows:
if *uplo* = 'U', $ab(kd+1+i-j,j) = A(i,j)$ for $\max(1,j-kd) \leq i \leq j$;
if *uplo* = 'L', $ab(1+i-j,j) = A(i,j)$ for $j \leq i \leq \min(n,j+kd)$.

ldab INTEGER.
The leading dimension of the array *ab*. $ldab \geq kd+1$.

x REAL for slatbs
DOUBLE PRECISION for dlatbs
COMPLEX for clatbs
COMPLEX*16 for zlatbs.
Array, DIMENSION (*n*).
On entry, the right hand side *b* of the triangular system.

cnorm REAL for slatbs/clatbs
DOUBLE PRECISION for dlatbs/zlatbs.
Array, DIMENSION (*n*).
If *normin* = 'Y', *cnorm* is an input argument and *cnorm*(*j*) contains the norm of the off-diagonal part of the *j*-th column of *A*. If *trans* = 'N', *cnorm*(*j*) must be greater than or equal to the infinity-norm, and if *trans* = 'T' or 'C', *cnorm*(*j*) must be greater than or equal to the 1-norm.

Output Parameters

scale REAL for slatbs/clatbs
DOUBLE PRECISION for dlatbs/zlatbs.
The scaling factor *s* for the triangular system as described above.
If *scale* = 0, the matrix *A* is singular or badly scaled, and the vector *x* is an exact or approximate solution to A*x* = 0.

cnorm If *normin* = 'N', *cnorm* is an output argument and *cnorm*(*j*) returns the 1-norm of the off-diagonal part of the *j*-th column of *A*.

info INTEGER.
= 0: successful exit
< 0: if *info* = -*k*, the *k*-th argument had an illegal value

?latdf

Uses the LU factorization of the n-by-n matrix computed by ?getc2 and computes a contribution to the reciprocal Dif-estimate.

Syntax

```
call slatdf ( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
call dlatdf ( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
```

```
call clatdf ( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
call zlatdf ( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
```

Description

The routine `?latdf` uses the *LU* factorization of the *n*-by-*n* matrix *Z* computed by `?getc2` and computes a contribution to the reciprocal
Dif-estimate by solving $Zx = b$ for *x*, and choosing the right-hand side *b* such that the norm of *x* is as large as possible. On entry *rhs* = *b* holds the contribution from earlier solved sub-systems, and on return *rhs* = *x*.

The factorization of *Z* returned by `?getc2` has the form $Z = P L U Q$, where *P* and *Q* are permutation matrices. *L* is lower triangular with unit diagonal elements and *U* is upper triangular.

Input Parameters

ijob — `INTEGER`.
ijob = 2: First compute an approximative null-vector *e* of *Z* using `?gecon`, *e* is normalized, and solve for
$Zx = \pm e - f$ with the sign giving the greater value of 2-norm(*x*). This option is about 5 times as expensive as default.
ijob $\neq 2$ (default): Local look ahead strategy where all entries of the right-hand side *b* is chosen as either +1 or -1 .

n — `INTEGER`.
The number of columns of the matrix *Z*.

z — `REAL` for `slatdf/clatdf`
`DOUBLE PRECISION` for `dlatdf/zlatdf`.
Array, `DIMENSION` (*ldz, n*)
On entry, the *LU* part of the factorization of the *n*-by-*n* matrix *Z* computed by `?getc2`: $Z = P L U Q$.

ldz — `INTEGER`.
The leading dimension of the array *z*. *lda* $\geq$ max(1, *n*).

rhs — `REAL` for `slatdf/clatdf`
`DOUBLE PRECISION` for `dlatdf/zlatdf`.
Array, `DIMENSION` (*n*).
On entry, *rhs* contains contributions from other subsystems.

rdsum — `REAL` for `slatdf/clatdf`
`DOUBLE PRECISION` for `dlatdf/zlatdf`.
On entry, the sum of squares of computed contributions to the *D*if-estimate

under computation by `?tgsyl`, where the scaling factor *rdscal* has been factored out.
If *trans* = 'T' , *rdsum* is not touched.
Note that *rdsum* only makes sense when `?tgsy2` is called by `?tgsyl`.

rdscal — `REAL` for `slatdf/clatdf`
`DOUBLE PRECISION` for `dlatdf/zlatdf`.
On entry, scaling factor used to prevent overflow in *rdsum*. If *trans* = `T`′, *rdscal* is not touched.
Note that *rdscal* only makes sense when `?tgsy2` is called by `?tgsyl`.

ipiv — `INTEGER`.
Array, `DIMENSION` (*n*).
The pivot indices; for $1 \le i \le n$, row *i* of the matrix has been interchanged with row *ipiv*(*i*).

jpiv — `INTEGER`.
Array, `DIMENSION` (*n*).
The pivot indices; for $1 \le j \le n$, column *j* of the matrix has been interchanged with column *jpiv*(*j*).

Output Parameters

rhs — On exit, *rhs* contains the solution of the subsystem with entries according to the value of *ijob*.

rdsum — On exit, the corresponding sum of squares updated with the contributions from the current sub-system.
If *trans* = 'T' , *rdsum* is not touched.

rdscal — On exit, *rdscal* is updated with respect to the current contributions in *rdsum*.
If *trans* = 'T', *rdscal* is not touched.

?latps

Solves a triangular system of equations with the matrix held in packed storage.

Syntax

```
call slatps (uplo, trans, diag, normin, n, ap, x, scale, cnorm, info)
call dlatps (uplo, trans, diag, normin, n, ap, x, scale, cnorm, info)
```

```
call clatps (uplo, trans, diag, normin, n, ap, x, scale, cnorm, info)
call zlatps (uplo, trans, diag, normin, n, ap, x, scale, cnorm, info)
```

Description

The routine `?latps` solves one of the triangular systems

$Ax = s\,b$ or $A^T x = s\,b$ or $A^H x = s\,b$ (for complex flavors)

with scaling to prevent overflow, where A is an upper or lower triangular matrix stored in packed form. Here A^T denotes the transpose of A, A^H denotes the conjugate transpose of A, x and b are n-element vectors, and s is a scaling factor, usually less than or equal to 1, chosen so that the components of x will be less than the overflow threshold. If the unscaled problem will not cause overflow, the Level 2 BLAS routine `?tpsv` is called. If the matrix A is singular ($A(j, j) = 0$ for some j), then s is set to 0 and a non-trivial solution to $Ax = 0$ is returned.

Input Parameters

uplo — `CHARACTER*1`.
Specifies whether the matrix A is upper or lower triangular.
= 'U': Upper triangular
= 'L': Lower triangular

trans — `CHARACTER*1`.
Specifies the operation applied to A.
= 'N': Solve $Ax = s\,b$ (no transpose)
= 'T': Solve $A^T x = s\,b$ (transpose)
= 'C': Solve $A^H x = s\,b$ (conjugate transpose)

diag — `CHARACTER*1`.
Specifies whether or not the matrix A is unit triangular.
= 'N': Non-unit triangular
= 'U': Unit triangular

normin — `CHARACTER*1`.
Specifies whether *cnorm* has been set or not.
= 'Y': *cnorm* contains the column norms on entry;
= 'N': *cnorm* is not set on entry. On exit, the norms will be computed and stored in *cnorm*.

n — `INTEGER`.
The order of the matrix A. $n \geq 0$.

ap — `REAL` for `slatps`
`DOUBLE PRECISION` for `dlatps`
`COMPLEX` for `clatps`
`COMPLEX*16` for `zlatps`.
Array, `DIMENSION` ($n(n+1)/2$). The upper or lower triangular matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array *ap* as follows:
if *uplo* = 'U', $ap(i + (j-1)j/2) = A(i,j)$ for $1 \le i \le j$;
if *uplo* = 'L', $ap(i + (j-1)(2n-j)/2) = A(i,j)$ for $j \le i \le n$.

x — `REAL` for `slatps`
`DOUBLE PRECISION` for `dlatps`
`COMPLEX` for `clatps`
`COMPLEX*16` for `zlatps`.
Array, `DIMENSION` (n)
On entry, the right hand side b of the triangular system.

cnorm — `REAL` for `slatps/clatps`
`DOUBLE PRECISION` for `dlatps/zlatps`.
Array, `DIMENSION` (n).
If *normin* = 'Y', *cnorm* is an input argument and *cnorm*(j) contains the norm of the off-diagonal part of the j-th column of A. If *trans* = 'N', *cnorm*(j) must be greater than or equal to the infinity-norm, and if *trans* = 'T' or 'C', *cnorm*(j) must be greater than or equal to the 1-norm.

Output Parameters

x — On exit, *x* is overwritten by the solution vector x.

scale — `REAL` for `slatps/clatps`
`DOUBLE PRECISION` for `dlatps/zlatps`.
The scaling factor s for the triangular system as described above.
If *scale* = 0, the matrix A is singular or badly scaled, and the vector x is an exact or approximate solution to
$Ax = 0$.

cnorm — If *normin* = 'N', *cnorm* is an output argument and *cnorm*(j) returns the 1-norm of the off-diagonal part of the j-th column of A.

info — `INTEGER`.
= 0: successful exit
< 0: if *info* = $-k$, the k-th argument had an illegal value

?latrd

Reduces the first nb *rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.*

Syntax

```
call slatrd ( uplo, n, nb, a, lda, e, tau, w, ldw )
call dlatrd ( uplo, n, nb, a, lda, e, tau, w, ldw )
call clatrd ( uplo, n, nb, a, lda, e, tau, w, ldw )
call zlatrd ( uplo, n, nb, a, lda, e, tau, w, ldw )
```

Description

The routine `?latrd` reduces *nb* rows and columns of a real symmetric or complex Hermitian matrix *A* to symmetric/Hermitian tridiagonal form by an orthogonal/unitary similarity transformation $Q' A Q$, and returns the matrices *V* and *W* which are needed to apply the transformation to the unreduced part of *A*.
If *uplo* = 'U', `?latrd` reduces the last *nb* rows and columns of a matrix, of which the upper triangle is supplied;
if *uplo* = 'L', `?latrd` reduces the first *nb* rows and columns of a matrix, of which the lower triangle is supplied.

This is an auxiliary routine called by `?sytrd/?hetrd`.

Input Parameters

uplo — `CHARACTER`
Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix *A* is stored:
= 'U': Upper triangular
= 'L': Lower triangular

n — `INTEGER`.
The order of the matrix *A*.

nb — `INTEGER`.
The number of rows and columns to be reduced.

a — REAL for slatrd
DOUBLE PRECISION for dlatrd
COMPLEX for clatrd
COMPLEX*16 for zlatrd.
Array, DIMENSION (*lda*, *n*) .
On entry, the symmetric/Hermitian matrix *A*
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda — INTEGER.
The leading dimension of the array *a*. *lda* ≥ (1,*n*).

ldw — INTEGER.
The leading dimension of the output array *w*.
ldw ≥ max(1,*n*).

Output Parameters

a — On exit, if *uplo* = 'U', the last *nb* columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of *a*; the elements above the diagonal with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors;
if *uplo* = 'L', the first *nb* columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of *a*; the elements below the diagonal with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.

e — REAL for slatrd/clatrd
DOUBLE PRECISION for dlatrd/zlatrd.
If *uplo* = 'U', *e*(*n*-*nb*:*n*-1) contains the superdiagonal elements of the last *nb* columns of the reduced matrix;
if *uplo* = 'L', *e*(1:*nb*) contains the subdiagonal elements of the first *nb* columns of the reduced matrix.

tau — REAL for slatrd
DOUBLE PRECISION for dlatrd
COMPLEX for clatrd
COMPLEX*16 for zlatrd.

Array, `DIMENSION` (*lda*, *n*).
The scalar factors of the elementary reflectors, stored in *tau*(*n*-*nb*:*n*-1) if *uplo* = 'U', and in *tau*(1:*nb*) if *uplo* = 'L'.

w — `REAL` for `slatrd`
`DOUBLE PRECISION` for `dlatrd`
`COMPLEX` for `clatrd`
`COMPLEX*16` for `zlatrd`.
Array, `DIMENSION` (*lda*, *n*).
The *n*-by-*nb* matrix *W* required to update the unreduced part of *A*.

Application Notes

If *uplo* = 'U', the matrix *Q* is represented as a product of elementary reflectors

$Q = H(n) H(n-1) \ldots H(n-nb+1)$

Each *H(i)* has the form
$H(i) = I - tau*v*v'$
where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(*i*:*n*) = 0 and *v*(*i*-1) = 1; *v*(1:*i*-1) is stored on exit in *a*(1:*i*-1, *i*), and *tau* in *tau*(*i*-1).

If *uplo* = 'L', the matrix *Q* is represented as a product of elementary reflectors

$Q = H(1) H(2) \ldots H(nb)$

Each *H(i)* has the form
$H(i) = I - tau*v*v'$
where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1:*i*) = 0 and *v*(*i*+1) = 1; *v*(*i*+1:n) is stored on exit in *a*(*i*+1:*n*, i), and *tau* in *tau*(i).

The elements of the vectors *v* together form the *n*-by-*nb* matrix *V* which is needed, with *W*, to apply the transformation to the unreduced part of the matrix, using a symmetric/Hermitian rank-2k update of the form:
$A := A - VW' - WV'$.

The contents of *a* on exit are illustrated by the following examples with
n = 5 and *nb* = 2:

if *uplo* = 'U': if *uplo* = 'L':

$$\begin{bmatrix} a & a & a & v_4 & v_5 \\ & a & a & v_4 & v_5 \\ & & a & 1 & v_5 \\ & & & d & 1 \\ & & & & d \end{bmatrix} \qquad \begin{bmatrix} d & & & & \\ 1 & d & & & \\ v_1 & 1 & a & & \\ v_1 & v_2 & a & a & \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

where d denotes a diagonal element of the reduced matrix, a denotes an element of the original matrix that is unchanged, and v_i denotes an element of the vector defining *H(i)*.

?latrs

Solves a triangular system of equations with the scale factor set to prevent overflow.

Syntax

```
call slatrs ( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
call dlatrs ( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
call clatrs ( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
call zlatrs ( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
```

Description

The routine solves one of the triangular systems

$Ax = s\,b$ or $A^T x = s\,b$ or $A^H x = s\,b$ (for complex flavors)

with scaling to prevent overflow. Here A is an upper or lower triangular matrix, A^T denotes the transpose of A, A^H denotes the conjugate transpose of A, x and b are n-element vectors, and s is a scaling factor, usually less than or equal to 1, chosen so that the components of x will be less than the overflow threshold. If the unscaled problem will not cause overflow, the Level 2 BLAS routine `?trsv` is called. If the matrix A is singular ($A(j,j) = 0$ for some j), then s is set to 0 and a non-trivial solution to $Ax = 0$ is returned.

Input Parameters

uplo

CHARACTER*1.
Specifies whether the matrix A is upper or lower triangular.
= 'U': Upper triangular
= 'L': Lower triangular

trans

CHARACTER*1.
Specifies the operation applied to A.
= 'N': Solve $Ax = s\ b$ (no transpose)
= 'T': Solve $A^T x = s\ b$ (transpose)
= 'C': Solve $A^H x = s\ b$ (conjugate transpose)

diag

CHARACTER*1.
Specifies whether or not the matrix A is unit triangular.
= 'N': Non-unit triangular
= 'U': Unit triangular

normin

CHARACTER*1.
Specifies whether *cnorm* has been set or not.
= 'Y': *cnorm* contains the column norms on entry;
= 'N': *cnorm* is not set on entry. On exit, the norms will be computed and stored in *cnorm*.

n

INTEGER.
The order of the matrix A. $n \geq 0$

a

REAL for slatrs
DOUBLE PRECISION for dlatrs
COMPLEX for clatrs
COMPLEX*16 for zlatrs.
Array, DIMENSION (*lda*, *n*). Contains the triangular matrix A. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of the array *a* contains the upper triangular matrix, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the array *a* contains the lower triangular matrix, and the strictly upper triangular part of *a* is not referenced. If *diag* = 'U', the diagonal elements of *a* are also not referenced and are assumed to be 1.

lda

INTEGER.
The leading dimension of the array *a*. $lda \geq \max(1, n)$.

x — REAL for slatrs
DOUBLE PRECISION for dlatrs
COMPLEX for clatrs
COMPLEX*16 for zlatrs.
Array, DIMENSION (*n*). On entry, the right hand side *b* of the triangular system.

cnorm — REAL for slatrs/clatrs)
DOUBLE PRECISION for dlatrs/zlatrs.
Array, DIMENSION (*n*). If *normin* = 'Y', *cnorm* is an input argument and *cnorm* (*j*) contains the norm of the off-diagonal part of the *j*-th column of *A*. If *trans* = 'N', *cnorm* (*j*) must be greater than or equal to the infinity-norm, and if *trans* = 'T' or 'C', *cnorm*(*j*) must be greater than or equal to the 1-norm.

Output Parameters

x — On exit, *x* is overwritten by the solution vector *x*.

scale — REAL for slatrs/clatrs)
DOUBLE PRECISION for dlatrs/zlatrs.
Array, DIMENSION (*lda*, *n*). The scaling factor *s* for the triangular system as described above.
If *scale* = 0, the matrix *A* is singular or badly scaled, and the vector *x* is an exact or approximate solution to $Ax = 0$.

cnorm — If *normin* = '*N*', *cnorm* is an output argument and *cnorm*(*j*) returns the 1-norm of the off-diagonal part of the *j*-th column of *A*.

info — INTEGER.
= 0: successful exit
< 0: if *info* = -*k*, the *k*-th argument had an illegal value

Application Notes

A rough bound on *x* is computed; if that is less than overflow, ?trsv is called, otherwise, specific code is used which checks for possible overflow or divide-by-zero at every operation.

A columnwise scheme is used for solving $Ax = b$. The basic algorithm if *A* is lower triangular is

```
x[1:n] := b[1:n]
for j = 1, ..., n
x(j) := x(j) / A(j,j)
x[j+1:n] := x[j+1:n] - x(j)*A[j+1:n,j]
end
```

Define bounds on the components of x after j iterations of the loop:
$M(j)$ = bound on $x[1:j]$
$G(j)$ = bound on $x[j+1:n]$
Initially, let $M(0) = 0$ and $G(0) = \max\{x(i), i=1,...,n\}$.

Then for iteration j+1 we have
$M(j+1) \le G(j) / | A(j+1,j+1) |$
$G(j+1) \le G(j) + M(j+1)*| A[j+2:n,j+1] |$
$\le G(j) (1 + cnorm(j+1) / | A(j+1,j+1) |$,

where $cnorm(j+1)$ is greater than or equal to the infinity-norm of column j+1 of A, not counting the diagonal. Hence

$$G(j) \le G(0) \prod_{1 \le i \le j} (1 + cnorm(i)/|A(i,i)|)$$

and

$$|x(j)| \le (G(0)/|A(j,j)|) \prod_{1 \le i \le j} (1 + cnorm(i)/|A(i,i)|)$$

Since $|x(j)| \le M(j)$, we use the Level 2 BLAS routine `?trsv` if the reciprocal of the largest $M(j)$, $j=1,..,n$, is larger than
max(*underflow*, 1/*overflow*).
The bound on $x(j)$ is also used to determine when a step in the columnwise method can be performed without fear of overflow. If the computed bound is greater than a large constant, x is scaled to prevent overflow, but if the bound overflows, x is set to 0, $x(j)$ to 1, and scale to 0, and a non-trivial solution to $Ax = 0$ is found.

Similarly, a row-wise scheme is used to solve $A^T x = b$ or $A^H x = b$. The basic algorithm for A upper triangular is

for $j = 1, ..., n$
$x(j) := (b(j) - A[1:j-1,j]' \; x[1:j-1]) / A(j,j)$
end

We simultaneously compute two bounds
$G(j)$ = bound on ($b(i) - A[1:i-1,i]'*x[1:i-1]$), $1 \le i \le j$
$M(j)$ = bound on $x(i)$, $1 \le i \le j$

The initial values are $G(0) = 0$, $M(0) = \max\{b(i), i=1,..,n\}$, and we add the constraint $G(j) \geq G(j-1)$ and $M(j) \geq M(j-1)$ for $j \geq 1$.
Then the bound on $x(j)$ is

$$M(j) \leq M(j-1) * (1 + cnorm(j)) / |A(j,j)|$$

$$\leq M(0) \prod_{1 \leq i \leq j} (1 + cnorm(i)/|A(i,i)|)$$

and we can safely call `?trsv` if $1/M(n)$ and $1/G(n)$ are both greater than max(*underflow*, 1/*overflow*).

?latrz

Factors an upper trapezoidal matrix by means of orthogonal/unitary transformations.

Syntax

```
call slatrz ( m, n, l, a, lda, tau, work )
call dlatrz ( m, n, l, a, lda, tau, work )
call clatrz ( m, n, l, a, lda, tau, work )
call zlatrz ( m, n, l, a, lda, tau, work )
```

Description

The routine `?latrz` factors the m-by-$(m+l)$ real/complex upper trapezoidal matrix
$[\ A1\ \ A2\ ] = [\ A(1:m,1:m)\ \ \ A(1:m, n-l+1:n)\ ]$

as $(\ R\ \ \ 0\ )*Z$, by means of orthogonal/unitary transformations. Z is an $(m+l)$-by-$(m+l)$ orthogonal/unitary matrix and R and $A1$ are m-by-m upper triangular matrices.

Input Parameters

m — `INTEGER`.
The number of rows of the matrix A. $m \geq 0$.

n — `INTEGER`.
The number of columns of the matrix A. $n \geq 0$.

l
INTEGER.
The number of columns of the matrix *A* containing the meaningful part of the Householder vectors.
$n-m \geq l \geq 0$.

a
REAL for slatrz
DOUBLE PRECISION for dlatrz
COMPLEX for clatrz
COMPLEX*16 for zlatrz.
Array, DIMENSION (*lda*, *n*).
On entry, the leading *m*-by-*n* upper trapezoidal part of the array *a* must contain the matrix to be factorized.

lda
INTEGER.
The leading dimension of the array *a*. $lda \geq \max(1,m)$.

work
REAL for slatrz
DOUBLE PRECISION for dlatrz
COMPLEX for clatrz
COMPLEX*16 for zlatrz.
Workspace array, DIMENSION (*m*).

Output Parameters

a
On exit, the leading *m*-by-*m* upper triangular part of *a* contains the upper triangular matrix *R*, and elements $n-l+1$ to *n* of the first *m* rows of *a*, with the array *tau*, represent the orthogonal/unitary matrix *Z* as a product of *m* elementary reflectors.

tau
REAL for slatrz
DOUBLE PRECISION for dlatrz
COMPLEX for clatrz
COMPLEX*16 for zlatrz.
Array, DIMENSION (*m*). The scalar factors of the elementary reflectors.

Application Notes

The factorization is obtained by Householder's method. The *k*-th transformation matrix, $Z(k)$, which is used to introduce zeros into the
$(m - k + 1)$-th row of *A*, is given in the form

$$Z(k) = \begin{bmatrix} I & 0 \\ 0 & T(k) \end{bmatrix}$$

where

$$T(k) = I - tau * u(k) * u(k)', \quad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

tau is a scalar and *z(k)* is an *l*-element vector. *tau* and *z(k)* are chosen to annihilate the elements of the *k*-th row of *A*2.
The scalar *tau* is returned in the *k*-th element of `tau` and the vector *u(k)* in the *k*-th row of *A*2, such that the elements of *z(k)* are in $a(k, l+1), \ldots, a(k, n)$. The elements of *R* are returned in the upper triangular part of *A*1.
Z is given by

$$Z = Z(1)\, Z(2) \ldots Z(m).$$

?lauu2

Computes the product UU^H or L^HL, where U and L are upper or lower triangular matrices (unblocked algorithm).

Syntax

```
call slauu2 ( uplo, n, a, lda, info )
call dlauu2 ( uplo, n, a, lda, info )
call clauu2 ( uplo, n, a, lda, info )
call zlauu2 ( uplo, n, a, lda, info )
```

Description

The routine ?lauu2 computes the product UU' or $L'L$, where the triangular factor U or L is stored in the upper or lower triangular part of the array *a*.
If *uplo* = 'U' or 'u' , then the upper triangle of the result is stored, overwriting the factor U in *a*.
If *uplo* = 'L' or 'l' , then the lower triangle of the result is stored, overwriting the factor L in *a*.

This is the unblocked form of the algorithm, calling Level 2 BLAS.

Input Parameters

uplo CHARACTER*1.
Specifies whether the triangular factor stored in the array *a* is upper or lower triangular:
= 'U': Upper triangular
= 'L': Lower triangular

n INTEGER.
The order of the triangular factor U or L. $n \geq 0$.

a REAL for slauu2
DOUBLE PRECISION for dlauu2
COMPLEX for clauu2
COMPLEX*16 for zlauu2.
Array, DIMENSION (*lda*, *n*).On entry, the triangular factor U or L.

lda INTEGER.
The leading dimension of the array *a*. $lda \geq \max(1,n)$.

Output Parameters

a On exit, if *uplo* = 'U', the upper triangle of *a* is overwritten with the upper triangle of the product UU' ; if *uplo* = 'L', the lower triangle of *a* is overwritten with the lower triangle of the product $L'L$.

info INTEGER.
= 0: successful exit
< 0: if *info* = -*k*, the *k*-th argument had an illegal value

?lauum

Computes the product UU^H or L^HL, where U and L are upper or lower triangular matrices.

Syntax

```
call slauum ( uplo, n, a, lda, info )
call dlauum ( uplo, n, a, lda, info )
call clauum ( uplo, n, a, lda, info )
call zlauum ( uplo, n, a, lda, info )
```

Description

The routine `?lauum` computes the product *UU'* or *L'L*, where the triangular factor *U* or *L* is stored in the upper or lower triangular part of the array *a*.

If *uplo* = 'U' or 'u', then the upper triangle of the result is stored, overwriting the factor *U* in *a*.
If *uplo* = 'L' or 'l', then the lower triangle of the result is stored, overwriting the factor *L* in *a*.

This is the blocked form of the algorithm, calling Level 3 BLAS.

Input Parameters

uplo — `CHARACTER*1`.
Specifies whether the triangular factor stored in the array *a* is upper or lower triangular:
= 'U': Upper triangular
= 'L': Lower triangular

n — `INTEGER`.
The order of the triangular factor *U* or *L*. $n \geq 0$.

a — `REAL` for `slauum`
`DOUBLE PRECISION` for `dlauum`
`COMPLEX` for `clauum`
`COMPLEX*16` for `zlauum`.
Array, `DIMENSION` (*lda*, *n*). On entry, the triangular factor *U* or *L*.

lda — `INTEGER`.
The leading dimension of the array *a*. $lda \geq \max(1,n)$.

Output Parameters

a — On exit, if *uplo* = 'U', the upper triangle of *a* is overwritten with the upper triangle of the product UU' ; if *uplo* = 'L', the lower triangle of *a* is overwritten with the lower triangle of the product $L'L$.

info — INTEGER.
= 0: successful exit
< 0: if *info* = -*k*, the *k*-th argument had an illegal value

?org2l/?ung2l

Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by `?geqlf` *(unblocked algorithm).*

Syntax

```
call sorg2l ( m, n, k, a, lda, tau, work, info )
call dorg2l ( m, n, k, a, lda, tau, work, info )
call cung2l ( m, n, k, a, lda, tau, work, info )
call zung2l ( m, n, k, a, lda, tau, work, info )
```

Description

The routine `?org2l/?ung2l` generates an *m*-by-*n* real/complex matrix Q with orthonormal columns, which is defined as the last *n* columns of a product of *k* elementary reflectors of order *m*:

$Q = H(k) \ldots H(2)\, H(1)$ as returned by `?geqlf`.

Input Parameters

m — INTEGER.
The number of rows of the matrix Q. $m \geq 0$.

n — INTEGER.
The number of columns of the matrix Q. $m \geq n \geq 0$.

k	`INTEGER`. The number of elementary reflectors whose product defines the matrix *Q*. $n \geq k \geq 0$.
a	`REAL` for `sorg2l` `DOUBLE PRECISION` for `dorg2l` `COMPLEX` for `cung2l` `COMPLEX*16` for `zung2l`. Array, `DIMENSION (lda,n)`. On entry, the ($n-k+i$)-th column must contain the vector which defines the elementary reflector *H(i)*, for $i = 1,2,...,k$, as returned by `?geqlf` in the last *k* columns of its array argument *a*.
lda	`INTEGER`. The first dimension of the array *a*. $lda \geq \max(1,m)$.
tau	`REAL` for `sorg2l` `DOUBLE PRECISION` for `dorg2l` `COMPLEX` for `cung2l` `COMPLEX*16` for `zung2l`. Array, `DIMENSION (k)`. *tau*(*i*) must contain the scalar factor of the elementary reflector *H(i)*, as returned by `?geqlf`.
work	`REAL` for `sorg2l` `DOUBLE PRECISION` for `dorg2l` `COMPLEX` for `cung2l` `COMPLEX*16` for `zung2l`. Workspace array, `DIMENSION` (*n*).

Output Parameters

a	On exit, the *m*-by-*n* matrix *Q*.
info	`INTEGER`. = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument has an illegal value

?org2r/?ung2r

Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by `?geqrf` (unblocked algorithm).

Syntax

```
call sorg2r ( m, n, k, a, lda, tau, work, info )
call dorg2r ( m, n, k, a, lda, tau, work, info )
call cung2r ( m, n, k, a, lda, tau, work, info )
call zung2r ( m, n, k, a, lda, tau, work, info )
```

Description

The routine `?org2r/?ung2r` generates an m-by-n real/complex matrix Q 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 `?geqrf`.

Input Parameters

m — `INTEGER`.
The number of rows of the matrix Q. $m \geq 0$.

n — `INTEGER`.
The number of columns of the matrix Q. $m \geq n \geq 0$.

k — `INTEGER`.
The number of elementary reflectors whose product defines the matrix Q. $n \geq k \geq 0$.

a — `REAL` for `sorg2r`
`DOUBLE PRECISION` for `dorg2r`
`COMPLEX` for `cung2r`
`COMPLEX*16` for `zung2r`.
Array, `DIMENSION` (*lda*, *n*).

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 `?geqrf` in the first k columns of its array argument `a`.

lda — `INTEGER`.
The first `DIMENSION` of the array `a`. $lda \geq \max(1,m)$.

tau — `REAL` for `sorg2r`
`DOUBLE PRECISION` for `dorg2r`
`COMPLEX` for `cung2r`
`COMPLEX*16` for `zung2r`.
Array, `DIMENSION` (k).
`tau`(i) must contain the scalar factor of the elementary reflector $H(i)$, as returned by `?geqrf`.

work — `REAL` for `sorg2r`
`DOUBLE PRECISION` for `dorg2r`
`COMPLEX` for `cung2r`
`COMPLEX*16` for `zung2r`.
Workspace array, `DIMENSION` (n).

Output Parameters

a — On exit, the m-by-n matrix Q.

info — `INTEGER`.
= 0: successful exit
< 0: if $info = -i$, the i-th argument has an illegal value

?orgl2/?ungl2

Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by `?gelqf` *(unblocked algorithm).*

Syntax

```
call sorgl2 ( m, n, k, a, lda, tau, work, info )
call dorgl2 ( m, n, k, a, lda, tau, work, info )
call cungl2 ( m, n, k, a, lda, tau, work, info )
call zungl2 ( m, n, k, a, lda, tau, work, info )
```

Description

The routine `?orgl2/?ungl2` generates a m-by-n real/complex matrix Q with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$$Q = H(k) \ldots H(2)\, H(1) \text{ or } Q = H(k)' \ldots H(2)'\, H(1)'$$

as returned by `?gelqf`.

Input Parameters

m — `INTEGER`.
The number of rows of the matrix Q. $m \geq 0$.

n — `INTEGER`.
The number of columns of the matrix Q. $n \geq m$.

k — `INTEGER`.
The number of elementary reflectors whose product defines the matrix Q. $m \geq k \geq 0$.

a — `REAL` for `sorgl2`
`DOUBLE PRECISION` for `dorgl2`
`COMPLEX` for `cungl2`
`COMPLEX*16` for `zungl2`.
Array, `DIMENSION` (*lda*, *n*). 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 `?gelqf` in the first *k* rows of its array argument *a*.

lda — `INTEGER`.
The first dimension of the array *a*. $lda \geq \max(1,m)$.

tau — `REAL` for `sorgl2`
`DOUBLE PRECISION` for `dorgl2`
`COMPLEX` for `cungl2`
`COMPLEX*16` for `zungl2`.
Array, `DIMENSION` (*k*).
tau(*i*) must contain the scalar factor of the elementary reflector H(*i*), as returned by `?gelqf`.

work — `REAL` for `sorgl2`
`DOUBLE PRECISION` for `dorgl2`
`COMPLEX` for `cungl2`
`COMPLEX*16` for `zungl2`.
Workspace array, `DIMENSION` (*m*).

Output Parameters

a On exit, the *m*-by-*n* matrix *Q*.

info INTEGER.
= 0: successful exit
< 0: if *info* = -*i*, the *i*-th argument has an illegal value.

?orgr2/?ungr2

Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by ?gerqf *(unblocked algorithm).*

Syntax

```
call sorgr2 ( m, n, k, a, lda, tau, work, info )
call dorgr2 ( m, n, k, a, lda, tau, work, info )
call cungr2 ( m, n, k, a, lda, tau, work, info )
call zungr2 ( m, n, k, a, lda, tau, work, info )
```

Description

The routine ?orgr2/?ungr2 generates an *m*-by-*n* real matrix *Q* with orthonormal rows, which is defined as the last *m* rows of a product of *k* elementary reflectors of order *n*
$Q = H(1)\,H(2)\ldots H(k)$ or $Q = H(1)'\,H(2)'\ldots H(k)'$
as returned by ?gerqf.

Input Parameters

m INTEGER. The number of rows of the matrix *Q*. $m \geq 0$.

n INTEGER.
The number of columns of the matrix *Q*. $n \geq m$.

k INTEGER.
The number of elementary reflectors whose product defines the matrix *Q*. $m \geq k \geq 0$.

a REAL for sorgr2
DOUBLE PRECISION for dorgr2
COMPLEX for cungr2

COMPLEX*16 for zungr2.
Array, DIMENSION (*lda*, *n*).On entry, the (*m*-*k*+*i*)-th row must contain the vector which defines the elementary reflector *H*(*i*), for *i* = 1,2,...,*k*, as returned by ?gerqf in the last *k* rows of its array argument *a*.

lda INTEGER.
The first dimension of the array *a*. $lda \geq \max(1,m)$.

tau REAL for sorgr2
DOUBLE PRECISION for dorgr2
COMPLEX for cungr2
COMPLEX*16 for zungr2.
Array, DIMENSION (*k*).*tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by ?gerqf.

work REAL for sorgr2
DOUBLE PRECISION for dorgr2
COMPLEX for cungr2
COMPLEX*16 for zungr2.
Workspace array, DIMENSION (*m*).

Output Parameters

a On exit, the *m*-by-*n* matrix *Q*.

info INTEGER.
= 0: successful exit
< 0: if *info* = -*i*, the *i*-th argument has an illegal value

?orm2l/?unm2l

Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by ?geqlf *(unblocked algorithm).*

Syntax

```
call sorm2l ( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dorm2l ( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunm2l ( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunm2l ( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Description

The routine `?orm2l/?unm2l` overwrites the general real/complex *m*-by-*n* matrix *C* with

$Q*C$ if *side* = 'L' and *trans* = 'N', or
$Q'*C$ if *side* = 'L' and *trans* = 'T' (for real flavors) or
trans = 'C' (for complex flavors), or
$C*Q$ if *side* = 'R' and *trans* = 'N', or
$C*Q'$ if *side* = 'R' and *trans* = 'T' (for real flavors) or
trans = 'C' (for complex flavors)

where *Q* is a real orthogonal or complex unitary matrix defined as the product of *k* elementary reflectors

$$Q = H(k) \dots H(2)\, H(1)$$

as returned by `?geqlf`. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side — `CHARACTER*1`.
= 'L': apply *Q* or *Q*' from the left
= 'R': apply *Q* or *Q*′ from the right

trans — `CHARACTER*1`.
= 'N': apply *Q* (No transpose)
= 'T': apply *Q*′ (Transpose, for real flavors)
= 'C': apply *Q*′ (Conjugate transpose, for complex flavors)

m — `INTEGER`.
The number of rows of the matrix *C*. $m \geq 0$.

n — `INTEGER`.
The number of columns of the matrix *C*. $n \geq 0$.

k — `INTEGER`.
The number of elementary reflectors whose product defines the matrix Q.
If *side* = 'L', $m \geq k \geq 0$;
if *side* = 'R', $n \geq k \geq 0$.

a — `REAL` for `sorm2l`
`DOUBLE PRECISION` for `dorm2l`
`COMPLEX` for `cunm2l`
`COMPLEX*16` for `zunm2l`.

	Array, `DIMENSION` (*lda*,*k*).The *i*-th column must contain the vector which defines the elementary reflector *H(i)*, for *i* = 1,2,...,*k*, as returned by `?geqlf` in the last *k* columns of its array argument *a*. The array *a* is modified by the routine but restored on exit.
lda	`INTEGER`. The leading dimension of the array *a*. If *side* = 'L', *lda* ≥ max(1, *m*); if *side* = 'R', *lda* ≥ max(1, *n*).
tau	`REAL` for `sorm2l` `DOUBLE PRECISION` for `dorm2l` `COMPLEX` for `cunm2l` `COMPLEX*16` for `zunm2l`. Array, `DIMENSION` (*k*). *tau*(*i*) must contain the scalar factor of the elementary reflector H(*i*), as returned by `?geqlf`.
c	`REAL` for `sorm2l` `DOUBLE PRECISION` for `dorm2l` `COMPLEX` for `cunm2l` `COMPLEX*16` for `zunm2l`. Array, `DIMENSION` (*ldc*, *n*).On entry, the *m*-by-*n* matrix *C*.
ldc	`INTEGER`. The leading dimension of the array *C*. *ldc* ≥ max(1,*m*).
work	`REAL` for `sorm2l` `DOUBLE PRECISION` for `dorm2l` `COMPLEX` for `cunm2l` `COMPLEX*16` for `zunm2l`. Workspace array, `DIMENSION`: (*n*) if *side* = 'L', (*m*) if *side* = 'R'.

Output Parameters

c	On exit, *c* is overwritten by *QC* or *Q'C* or *CQ'* or *CQ*.
info	`INTEGER`. = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument had an illegal value

?orm2r/?unm2r

Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by `?geqrf` *(unblocked algorithm).*

Syntax

```
call sorm2r ( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dorm2r ( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunm2r ( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunm2r ( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Description

The routine `?orm2r/?unm2r` overwrites the general real/complex *m*-by-*n* matrix *C* with

$Q*C$ if *side* = 'L' and *trans* = 'N', or
$Q'*C$ if *side* = 'L' and *trans* = 'T' (for real flavors) or
trans = 'C' (for complex flavors), or
$C*Q$ if *side* = 'R' and *trans* = 'N', or
$C*Q'$ if *side* = 'R' and *trans* = 'T' (for real flavors) or
trans = 'C' (for complex flavors)

where *Q* is a real orthogonal or complex unitary matrix defined as the product of *k* elementary reflectors

$$Q = H(1)\, H(2)\, .\,.\,.\, H(k)$$

as returned by `?geqrf`. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side CHARACTER*1.
= 'L': apply *Q* or *Q*' from the left
= 'R': apply *Q* or *Q*' from the right

trans CHARACTER*1.
= 'N': apply *Q* (No transpose)
= 'T': apply *Q*' (Transpose, for real flavors)
= 'C': apply *Q*' (Conjugate transpose, for complex flavors)

m INTEGER.
The number of rows of the matrix *C*. $m \geq 0$.

n INTEGER.
The number of columns of the matrix *C*. $n \geq 0$.

k INTEGER.
The number of elementary reflectors whose product defines the matrix *Q*.
If *side* = 'L', $m \geq k \geq 0$;
if *side* = 'R', $n \geq k \geq 0$.

a REAL for sorm2r
DOUBLE PRECISION for dorm2r
COMPLEX for cunm2r
COMPLEX*16 for zunm2r.
Array, DIMENSION (*lda*,*k*).The *i*-th column must contain the vector which defines the elementary reflector *H(i)*, for *i* = 1,2,...,*k*, as returned by ?geqrf in the first *k* columns of its array argument *a*. The array *a* is modified by the routine but restored on exit.

lda INTEGER.
The leading dimension of the array *a*.
If *side* = 'L', $lda \geq \max(1, m)$;
if *side* = 'R', $lda \geq \max(1, n)$.

tau REAL for sorm2r
DOUBLE PRECISION for dorm2r
COMPLEX for cunm2r
COMPLEX*16 for zunm2r.
Array, DIMENSION (*k*).
tau(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by ?geqrf.

c REAL for sorm2r
DOUBLE PRECISION for dorm2r
COMPLEX for cunm2r
COMPLEX*16 for zunm2r.
Array, DIMENSION (*ldc*, *n*). On entry, the *m*-by-*n* matrix *C*.

ldc INTEGER.
The leading dimension of the array *C*. $ldc \geq \max(1,m)$.

work REAL for sorm2r
DOUBLE PRECISION for dorm2r
COMPLEX for cunm2r

`COMPLEX*16` for `zunm2r`.
Workspace array, `DIMENSION`
(*n*) if *side* = 'L',
(*m*) if *side* = 'R'.

Output Parameters

c	On exit, *c* is overwritten by *QC* or *Q'C* or *CQ'* or *CQ*.
info	`INTEGER`. = 0: successful exit < 0: if *info* = -i, the *i*-th argument had an illegal value

?orml2/?unml2

Multiplies a general matrix by the orthogonal/unitary matrix from a LQ factorization determined by `?gelqf` *(unblocked algorithm).*

Syntax

```
call sorml2 ( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dorml2 ( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunml2 ( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunml2 ( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Description

The routine `?orml2/?unml2` overwrites the general real/complex *m*-by-*n* matrix *C* with

$Q*C$ if *side* = 'L' and *trans* = 'N', or
$Q'*C$ if *side* = 'L' and *trans* = 'T' (for real flavors) or
trans = 'C' (for complex flavors), or
$C*Q$ if *side* = 'R' and *trans* = 'N', or
$C*Q'$ if *side* = 'R' and *trans* = 'T' (for real flavors) or
trans = 'C' (for complex flavors)

where *Q* is a real orthogonal or complex unitary matrix defined as the product of *k* elementary reflectors

$$Q = H(k) \ldots H(2)\, H(1) \quad \text{or} \quad Q = H(k)' \ldots H(2)'\, H(1)'$$

as returned by ?gelqf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side — CHARACTER*1.
= 'L': apply *Q* or *Q′* from the left
= 'R': apply *Q* or *Q′* from the right

trans — CHARACTER*1.
= 'N': apply *Q* (No transpose)
= 'T': apply *Q′* (Transpose, for real flavors)
= 'C': apply *Q′* (Conjugate transpose, for complex flavors)

m — INTEGER.
The number of rows of the matrix *C*. *m* ≥ 0.

n — INTEGER.
The number of columns of the matrix *C*. *n* ≥ 0.

k — INTEGER.
The number of elementary reflectors whose product defines the matrix *Q*.
If *side* = 'L', *m* ≥ *k* ≥ 0;
if *side* = 'R', *n* ≥ *k* ≥ 0.

a — REAL for sorml2
DOUBLE PRECISION for dorml2
COMPLEX for cunml2
COMPLEX*16 for zunml2.
Array, DIMENSION
(*lda*, *m*) if *side* = 'L',
(*lda*, *n*) if *side* = 'R'
The *i*-th row must contain the vector which defines the elementary reflector *H(i)*, for *i* = 1,2,...,*k*, as returned by ?gelqf in the first *k* rows of its array argument *a*. The array *a* is modified by the routine but restored on exit.

lda — INTEGER.
The leading dimension of the array *a*. *lda* ≥ max(1,*k*).

tau — REAL for sorml2
DOUBLE PRECISION for dorml2
COMPLEX for cunml2
COMPLEX*16 for zunml2.

Array, DIMENSION (k).
tau(i) must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?gelqf.

c — REAL for sorml2
DOUBLE PRECISION for dorml2
COMPLEX for cunml2
COMPLEX*16 for zunml2.
Array, DIMENSION (ldc, n)
On entry, the m-by-n matrix C.

ldc — INTEGER.
The leading dimension of the array c. ldc $\geq$ max(1,m).

work — REAL for sorml2
DOUBLE PRECISION for dorml2
COMPLEX for cunml2
COMPLEX*16 for zunml2.
Workspace array, DIMENSION
(n) if side = 'L',
(m) if side = 'R'

Output Parameters

c — On exit, c is overwritten by QC or $Q'C$ or CQ' or CQ.

info — INTEGER.
= 0: successful exit
< 0: if info = -i, the i-th argument had an illegal value

?ormr2/?unmr2

Multiplies a general matrix by the orthogonal/unitary matrix from a RQ factorization determined by ?gerqf *(unblocked algorithm).*

Syntax

```
call sormr2 ( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dormr2 ( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunmr2 ( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

```
call zunmr2 ( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Description

The routine `?ormr2/?unmr2` overwrites the general real/complex *m*-by-*n* matrix *C* with

$Q*C$ if *side* = 'L' and *trans* = 'N', or
$Q'*C$ if *side* = 'L' and *trans* = 'T' (for real flavors) or
trans = 'C' (for complex flavors), or
$C*Q$ if *side* = 'R' and *trans* = 'N', or
$C*Q'$ if *side* = 'R' and *trans* = 'T' (for real flavors) or
trans = 'C' (for complex flavors)

where *Q* is a real orthogonal or complex unitary matrix defined as the product of *k* elementary reflectors

$$Q = H(1)\,H(2)\,...\,H(k) \quad \text{or} \quad Q = H(1)'\,H(2)'\,...\,H(k)'$$

as returned by `?gerqf`. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side — `CHARACTER*1`.
= 'L': apply *Q* or *Q'* from the left
= 'R': apply *Q* or *Q'* from the right

trans — `CHARACTER*1`.
= 'N': apply *Q* (No transpose)
= 'T': apply *Q'* (Transpose, for real flavors)
= 'C': apply *Q'* (Conjugate transpose, for complex flavors)

m — `INTEGER`.
The number of rows of the matrix *C*. $m \geq 0$.

n — `INTEGER`.
The number of columns of the matrix *C*. $n \geq 0$.

k — `INTEGER`.
The number of elementary reflectors whose product defines the matrix *Q*.
If *side* = 'L', $m \geq k \geq 0$;
if *side* = 'R', $n \geq k \geq 0$.

a — `REAL` for `sormr2`
`DOUBLE PRECISION` for `dormr2`
`COMPLEX` for `cunmr2`

	COMPLEX*16 for zunmr2. Array, DIMENSION (*lda*, *m*) if *side* = 'L', (*lda*, *n*) if *side* = 'R' The *i*-th row must contain the vector which defines the elementary reflector *H(i)*, for *i* = 1,2,...,*k*, as returned by ?gerqf in the last *k* rows of its array argument *a*. The array *a* is modified by the routine but restored on exit.
lda	INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*k*).
tau	REAL for sormr2 DOUBLE PRECISION for dormr2 COMPLEX for cunmr2 COMPLEX*16 for zunmr2. Array, DIMENSION (*k*). *tau*(*i*) must contain the scalar factor of the elementary reflector H(*i*), as returned by ?gerqf.
c	REAL for sormr2 DOUBLE PRECISION for dormr2 COMPLEX for cunmr2 COMPLEX*16 for zunmr2. Array, DIMENSION (*ldc*, *n*). On entry, the *m*-by-*n* matrix *C*.
ldc	INTEGER. The leading dimension of the array *C*. *ldc* ≥ max(1,*m*).
work	REAL for sormr2 DOUBLE PRECISION for dormr2 COMPLEX for cunmr2 COMPLEX*16 for zunmr2. Workspace array, DIMENSION (*n*) if *side* = 'L', (*m*) if *side* = 'R'

Output Parameters

c	On exit, *c* is overwritten by *QC* or *Q'C* or *CQ'* or *CQ*.
info	INTEGER. = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument had an illegal value

?ormr3/?unmr3

Multiplies a general matrix by the orthogonal/unitary matrix from a RZ factorization determined by `?tzrzf` *(unblocked algorithm).*

Syntax

```
call sormr3 (side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info)
call dormr3 (side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info)
call cunmr3 (side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info)
call zunmr3 (side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info)
```

Description

The routine `?ormr3/?unmr3` overwrites the general real/complex *m*-by-*n* matrix *C* with

$Q*C$ if *side* = 'L' and *trans* = 'N', or
$Q'*C$ if *side* = 'L' and *trans* = 'T' (for real flavors) or
trans = 'C' (for complex flavors), or
$C*Q$ if *side* = 'R' and *trans* = 'N', or
$C*Q'$ if *side* = 'R' and *trans* = 'T' (for real flavors) or
trans = 'C' (for complex flavors)

where *Q* is a real orthogonal or complex unitary matrix defined as the product of *k* elementary reflectors

$$Q = H(1)\, H(2) \ldots H(k)$$

as returned by `?tzrzf`. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side — `CHARACTER*1`.
= 'L': apply *Q* or *Q'* from the left
= 'R': apply *Q* or *Q'* from the right

trans — `CHARACTER*1`.
= 'N': apply *Q* (No transpose)
= 'T': apply *Q'* (Transpose, for real flavors)
= 'C': apply *Q'* (Conjugate transpose, for complex flavors)

m — `INTEGER`.
The number of rows of the matrix *C*. $m \geq 0$.

n — `INTEGER`.
The number of columns of the matrix *C*. $n \geq 0$.

k — `INTEGER`.
The number of elementary reflectors whose product defines the matrix *Q*.
If *side* = 'L', $m \geq k \geq 0$;
if *side* = 'R', $n \geq k \geq 0$.

l — `INTEGER`.
The number of columns of the matrix *A* containing the meaningful part of the Householder reflectors.
If *side* = 'L', $m \geq l \geq 0$,
if *side* = 'R', $n \geq l \geq 0$.

a — `REAL` for `sormr3`
`DOUBLE PRECISION` for `dormr3`
`COMPLEX` for `cunmr3`
`COMPLEX*16` for `zunmr3`.
Array, `DIMENSION`
(*lda*, *m*) if *side* = 'L',
(*lda*, *n*) if *side* = 'R'
The *i*-th row must contain the vector which defines the elementary reflector *H(i)*, for *i* = 1,2,...,*k*, as returned by `?tzrzf` in the last *k* rows of its array argument *a*. The array *a* is modified by the routine but restored on exit.

lda — `INTEGER`.
The leading dimension of the array *a*. $lda \geq \max(1,k)$.

tau — `REAL` for `sormr3`
`DOUBLE PRECISION` for `dormr3`
`COMPLEX` for `cunmr3`
`COMPLEX*16` for `zunmr3`.
Array, `DIMENSION` (*k*).
tau(*i*) must contain the scalar factor of the elementary reflector H(*i*), as returned by `?tzrzf`.

c — `REAL` for `sormr3`
`DOUBLE PRECISION` for `dormr3`
`COMPLEX` for `cunmr3`

COMPLEX*16 for zunmr3.
Array, DIMENSION (*ldc*, *n*).
On entry, the *m*-by-*n* matrix *C*.

ldc INTEGER.
The leading dimension of the array c. *ldc* ≥ max(1,*m*).

work REAL for sormr3
DOUBLE PRECISION for dormr3
COMPLEX for cunmr3
COMPLEX*16 for zunmr3.
Workspace array, DIMENSION
(*n*) if *side* = 'L',
(*m*) if *side* = 'R'.

Output Parameters

c On exit, c is overwritten by *QC* or *Q'C* or *CQ'* or *CQ*.

info INTEGER.
= 0: successful exit
< 0: if *info* = -*i*, the *i*-th argument had an illegal value

?pbtf2

Computes the Cholesky factorization of a symmetric/ Hermitian positive definite band matrix (unblocked algorithm).

Syntax

```
call spbtf2 ( uplo, n, kd, ab, ldab, info )
call dpbtf2 ( uplo, n, kd, ab, ldab, info )
call cpbtf2 ( uplo, n, kd, ab, ldab, info )
call zpbtf2 ( uplo, n, kd, ab, ldab, info )
```

Description

The routine computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite band matrix A. The factorization has the form
$A = U' U$, if `uplo` = 'U', or
$A = L L'$, if `uplo` = 'L',

where U is an upper triangular matrix, U' is the transpose of U, and L is lower triangular. This is the unblocked version of the algorithm, calling Level 2 BLAS.

Input Parameters

uplo — `CHARACTER*1`.
Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix A is stored:
= 'U': Upper triangular
= 'L': Lower triangular

n — `INTEGER`.
The order of the matrix A. $n \geq 0$.

kd — `INTEGER`.
The number of super-diagonals of the matrix A if *uplo* = 'U', or the number of sub-diagonals if *uplo* = 'L'.
$kd \geq 0$.

ab — `REAL` for `spbtf2`
`DOUBLE PRECISION` for `dpbtf2`
`COMPLEX` for `cpbtf2`
`COMPLEX*16` for `zpbtf2`.
Array, `DIMENSION` (*ldab*, *n*).
On entry, the upper or lower triangle of the symmetric/ Hermitian band matrix A, stored in the first kd+1 rows of the array. The j-th column of A is stored in the j-th column of the array *ab* as follows:
if *uplo* = 'U', $ab(kd+1+i-j,j) = A(i,j)$ for
$\max(1,j-kd) \leq i \leq j$;
if *uplo* = 'L', $ab(1+i-j,j) = A(i,j)$ for
$j \leq i \leq \min(n,j+kd)$.

ldab — `INTEGER`.
The leading dimension of the array *ab*. $ldab \geq kd+1$.

Output Parameters

ab — On exit, if *info* = 0, the triangular factor U or L from the Cholesky factorization $A = U' U$ or $A = L L'$ of the band matrix A, in the same storage format as A.

info — `INTEGER`.
= 0: successful exit
< 0: if *info* = -*k*, the *k*-th argument had an illegal value
> 0: if *info* = *k*, the leading minor of order *k* is not positive definite, and the factorization could not be completed.

?potf2

Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (unblocked algorithm).

Syntax

```
call spotf2 ( uplo, n, a, lda, info )
call dpotf2 ( uplo, n, a, lda, info )
call cpotf2 ( uplo, n, a, lda, info )
call zpotf2 ( uplo, n, a, lda, info )
```

Description

The routine `?potf2` computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite matrix A. The factorization has the form
$A = U' U$, if *uplo* = 'U', or
$A = L L'$, if *uplo* = 'L',
where U is an upper triangular matrix and L is lower triangular.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

Input Parameters

uplo CHARACTER*1.
Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix *A* is stored.
= 'U': Upper triangular
= 'L': Lower triangular

n INTEGER.
The order of the matrix *A*. $n \geq 0$.

a REAL for spotf2
DOUBLE PRECISION or dpotf2
COMPLEX for cpotf2
COMPLEX*16 for zpotf2.
Array, DIMENSION (*lda*, *n*).
On entry, the symmetric/Hermitian matrix *A*.
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda INTEGER.
The leading dimension of the array *a*. *lda* ≥ max(1,*n*).

Output Parameters

a On exit, if *info* = 0, the factor *U* or *L* from the Cholesky factorization *A* = *U*′ *U* or *A* = *L L*'.

info INTEGER.
= 0: successful exit
< 0: if *info* = -*k*, the *k*-th argument had an illegal value
> 0: if *info* = *k*, the leading minor of order *k* is not positive definite, and the factorization could not be completed.

?ptts2

Solves a tridiagonal system of the form AX=B using the $L D L^H$ factorization computed by `?pttrf`.

Syntax

```
call sptts2 ( n, nrhs, d, e, b, ldb )
call dptts2 ( n, nrhs, d, e, b, ldb )
call cptts2 ( iuplo, n, nrhs, d, e, b, ldb )
call zptts2 ( iuplo, n, nrhs, d, e, b, ldb )
```

Description

The routine `?ptts2` solves a tridiagonal system of the form

$$A X = B$$

Real flavors `sptts2/dptts2` use the $L D L'$ factorization of A computed by `spttrf/dpttrf`, and complex flavors `cptts2/zptts2` use the $U'D U$ or $L D L'$ factorization of A computed by `cpttrf/zpttrf`.

D is a diagonal matrix specified in the vector *d*, U (or L) is a unit bidiagonal matrix whose superdiagonal (subdiagonal) is specified in the vector *e*, and X and B are *n*-by-*nrhs* matrices.

Input Parameters

iuplo — `INTEGER`. Used with complex flavors only.
Specifies the form of the factorization and whether the vector *e* is the superdiagonal of the upper bidiagonal factor U or the subdiagonal of the lower bidiagonal factor L.
= 1: $A = U' D U$, *e* is the superdiagonal of U;
= 0: $A = L D L'$, *e* is the subdiagonal of L

n — `INTEGER`.
The order of the tridiagonal matrix A. $n \geq 0$.

nrhs — `INTEGER`.
The number of right hand sides, that is, the number of columns of the matrix B. $nrhs \geq 0$.

d — `REAL` for `sptts2/cptts2`
`DOUBLE PRECISION` for `dptts2/zptts2`.
Array, `DIMENSION` (*n*).
The *n* diagonal elements of the diagonal matrix *D* from the factorization of *A*.

e — `REAL` for `sptts2`
`DOUBLE PRECISION` for `dptts2`
`COMPLEX` for `cptts2`
`COMPLEX*16` for `zptts2`.
Array, `DIMENSION` (*n*-1).
Contains the (*n*-1) subdiagonal elements of the unit bidiagonal factor *L* from the *LDL*' factorization of *A* (for real flavors, or for complex flavors when *iuplo* = 0).
For complex flavors when *iuplo* = 1, *e* contains the (*n*-1) superdiagonal elements of the unit bidiagonal factor *U* from the factorization *A* = *U'DU*.

b — `REAL` for `sptts2/cptts2`
`DOUBLE PRECISION` for `dptts2/zptts2`.
Array, `DIMENSION` (*ldb*, *nrhs*).
On entry, the right hand side vectors *B* for the system of linear equations.

ldb — `INTEGER`.
The leading dimension of the array *B*. *ldb* ≥ max(1,*n*).

Output Parameters

b — On exit, the solution vectors, *X*.

?rscl

Multiplies a vector by the reciprocal of a real scalar.

Syntax

```
call srscl ( n, sa, sx, incx )
call drscl ( n, sa, sx, incx )
call csrscl ( n, sa, sx, incx )
call zdrscl ( n, sa, sx, incx )
```

Description

The routine ?rscl multiplies an n-element real/complex vector x by the real scalar $1/a$. This is done without overflow or underflow as long as the final result x/a does not overflow or underflow.

Input Parameters

n	INTEGER. The number of components of the vector x.
sa	REAL for srscl/csrscl DOUBLE PRECISION for drscl/zdrscl. The scalar a which is used to divide each component of the vector x. sa must be ≥ 0, or the subroutine will divide by zero.
sx	REAL for srscl DOUBLE PRECISION for drscl COMPLEX for csrscl COMPLEX*16 for zdrscl. Array, DIMENSION $(1+(n-1)*\text{abs}(incx))$. The n-element vector x.
incx	INTEGER. The increment between successive values of the vector *sx*. If *incx* > 0, $sx(1) = x(1)$ and $sx(1+(i-1)*incx) = x(i)$, $1 < i \leq n$.

Output Parameters

sx	On exit, the result x/a.

?sygs2/?hegs2

Reduces a symmetric/Hermitian definite generalized eigenproblem to standard form, using the factorization results obtained from ?potrf *(unblocked algorithm).*

Syntax

```
call ssygs2 ( itype, uplo, n, a, lda, b, ldb, info )
call dsygs2 ( itype, uplo, n, a, lda, b, ldb, info )
```

```
call chegs2 ( itype, uplo, n, a, lda, b, ldb, info )
call zhegs2 ( itype, uplo, n, a, lda, b, ldb, info )
```

Description

The routine `?sygs2/?hegs2` reduces a real symmetric-definite or a complex Hermitian-definite generalized eigenproblem to standard form.
If *itype* = 1, the problem is

$Ax = \lambda Bx$,

and *A* is overwritten by inv(*U'*)**A**inv(*U*) or inv(*L*)**A**inv(*L'*).

If *itype* = 2 or 3, the problem is

$ABx = \lambda x$ or $B Ax = \lambda x$,

and *A* is overwritten by *UAU'* or *L'AL*. *B* must have been previously factorized as *U' U* or *L L'* by `?potrf`.

Input Parameters

itype — `INTEGER`.
= 1: compute inv(*U'*)**A**inv(*U*) or inv(*L*)**A**inv(*L'*);
= 2 or 3: compute *UAU'* or *L' AL*.

uplo — `CHARACTER`
Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix *A* is stored, and how *B* has been factorized.
= 'U': Upper triangular
= 'L': Lower triangular

n — `INTEGER`.
The order of the matrices *A* and *B*. $n \geq 0$.

a — `REAL` for `ssygs2`
`DOUBLE PRECISION` for `dsygs2`
`COMPLEX` for `chegs2`
`COMPLEX*16` for `zhegs2`.
Array, `DIMENSION` (*lda*, *n*).
On entry, the symmetric/Hermitian matrix *A*.
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda INTEGER.
The leading dimension of the array a. *lda* ≥ max(1,*n*).

b REAL for ssygs2
DOUBLE PRECISION for dsygs2
COMPLEX for chegs2
COMPLEX*16 for zhegs2.
Array, DIMENSION (*ldb*, *n*).
The triangular factor from the Cholesky factorization of *B* as returned by ?potrf.

ldb INTEGER.
The leading dimension of the array *B*. *ldb* ≥ max(1,*n*).

Output Parameters

a On exit, if *info* = 0, the transformed matrix, stored in the same format as *A*.

info INTEGER.
= 0: successful exit.
< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?sytd2/?hetd2

Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (unblocked algorithm).

Syntax

```
call ssytd2 ( uplo, n, a, lda, d, e, tau, info )
call dsytd2 ( uplo, n, a, lda, d, e, tau, info )
call chetd2 ( uplo, n, a, lda, d, e, tau, info )
call zhetd2 ( uplo, n, a, lda, d, e, tau, info )
```

Description

The routine ?sytd2/?hetd2 reduces a real symmetric/complex Hermitian matrix *A* to real symmetric tridiagonal form *T* by an orthogonal/unitary similarity transformation: $Q' AQ = T$.

Input Parameters

uplo — CHARACTER*1.
Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix *A* is stored:
= 'U': Upper triangular
= 'L': Lower triangular

n — INTEGER.
The order of the matrix *A*. *n* ≥ 0.

a — REAL for ssytd2
DOUBLE PRECISION for dsytd2
COMPLEX for chetd2
COMPLEX*16 for zhetd2.
Array, DIMENSION (*lda*, *n*).
On entry, the symmetric/Hermitian matrix *A*.
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda — INTEGER.
The leading dimension of the array *a*. *lda* ≥ max(1,*n*).

Output Parameters

a — On exit, if *uplo* = 'U', the diagonal and first superdiagonal of *a* are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first superdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors;
if *uplo* = 'L', the diagonal and first subdiagonal of *a* are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements below the first subdiagonal, with the array *tau,* represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.

d — REAL for ssytd2/chetd2
DOUBLE PRECISION for dsytd2/zhetd2.
Array, DIMENSION (*n*).
The diagonal elements of the tridiagonal matrix *T*:
d(*i*) = *a*(*i*,*i*).

e REAL for ssytd2/chetd2
DOUBLE PRECISION for dsytd2/zhetd2.
Array, DIMENSION (*n*-1).
The off-diagonal elements of the tridiagonal matrix *T*:
e(*i*) = *a*(*i*,*i*+1) if *uplo* = 'U',
e(*i*) = *a*(*i*+1,*i*) if *uplo* = 'L'.

tau REAL for ssytd2
DOUBLE PRECISION for dsytd2
COMPLEX for chetd2
COMPLEX*16 for zhetd2.
Array, DIMENSION (*n*-1).
The scalar factors of the elementary reflectors .

info INTEGER.
= 0: successful exit
< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?sytf2

Computes the factorization of a real/complex symmetric indefinite matrix, using the diagonal pivoting method (unblocked algorithm).

Syntax

```
call ssytf2 ( uplo, n, a, lda, ipiv, info )
call dsytf2 ( uplo, n, a, lda, ipiv, info )
call ñsytf2 ( uplo, n, a, lda, ipiv, info )
call zsytf2 ( uplo, n, a, lda, ipiv, info )
```

Description

The routine ?sytf2 computes the factorization of a real/complex symmetric matrix *A* using the Bunch-Kaufman diagonal pivoting method:

$$A = U D U' \text{ or } A = L D L'$$

where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, *U'* is the transpose of *U*, and *D* is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

Input Parameters

uplo — `CHARACTER*1`.
Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored
= 'U': Upper triangular
= 'L': Lower triangular

n — `INTEGER`.
The order of the matrix *A*. $n \geq 0$.

a — `REAL` for `ssytf2`
`DOUBLE PRECISION` for `dsytf2`
`COMPLEX` for `csytf2`
`COMPLEX*16` for `zsytf2`.
Array, `DIMENSION` (*lda*, *n*).
On entry, the symmetric matrix *A*.
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda — `INTEGER`.
The leading dimension of the array *a*. $lda \geq \max(1,n)$.

Output Parameters

a — On exit, the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L*.

ipiv — `INTEGER`.
Array, `DIMENSION` (*n*).
Details of the interchanges and the block structure of *D* If $ipiv(k) > 0$, then rows and columns *k* and $ipiv(k)$ were interchanged and $D(k,k)$ is a 1-by-1 diagonal block.
If *uplo* = 'U' and $ipiv(k) = ipiv(k-1) < 0$, then rows and columns k-1 and $-ipiv(k)$ were interchanged and $D(k-1:k,k-1:k)$ is a 2-by-2 diagonal block.
If *uplo* = 'L' and $ipiv(k) = ipiv(k+1) < 0$, then rows and columns k+1 and $-ipiv(k)$ were interchanged and $D(k:k+1,k:k+1)$ is a 2-by-2 diagonal block.

info INTEGER.
= 0: successful exit
< 0: if *info* = -*k*, the *k*-th argument had an illegal value
> 0: if *info* = *k*, *D(k,k)* is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?hetf2

Computes the factorization of a complex Hermitian matrix, using the diagonal pivoting method (unblocked algorithm).

Syntax

```
call chetf2 ( uplo, n, a, lda, ipiv, info )
call zhetf2 ( uplo, n, a, lda, ipiv, info )
```

Description

The routine computes the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method:

$A = U D U'$ or $A = L D L'$

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, U' is the conjugate transpose of U, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling Level 2 BLAS.

Input Parameters

uplo CHARACTER*1.
Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored:
= 'U': Upper triangular
= 'L': Lower triangular

n INTEGER.
The order of the matrix A. $n \geq 0$.

a — COMPLEX for chetf2
COMPLEX*16 for zhetf2.
Array, DIMENSION (*lda*, *n*).
On entry, the Hermitian matrix *A*.
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda — INTEGER.
The leading dimension of the array *a*. $lda \geq \max(1,n)$.

Output Parameters

a — On exit, the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L*.

ipiv — INTEGER.
Array, DIMENSION (*n*).
Details of the interchanges and the block structure of *D* If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) were interchanged and *D(k,k)* is a 1-by-1 diagonal block.
If *uplo* = 'U' and *ipiv*(*k*) = *ipiv*(*k*-1) < 0, then rows and columns *k*-1 and -*ipiv*(*k*) were interchanged and *D(k-1:k,k-1:k)* is a 2-by-2 diagonal block.
If *uplo* = 'L' and *ipiv*(*k*) = *ipiv*(*k*+1) < 0, then rows and columns *k*+1 and -*ipiv*(*k*) were interchanged and *D(k:k*+1,*k:k*+1) is a 2-by-2 diagonal block.

info — INTEGER.
= 0: successful exit
< 0: if *info* = -*k*, the *k*-th argument had an illegal value
> 0: if *info* = *k*, *D(k,k)* is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?tgex2

Swaps adjacent diagonal blocks in an upper (quasi) triangular matrix pair by an orthogonal/unitary equivalence transformation.

Syntax

```
call stgex2 ( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, n1,
    n2, work, lwork, info )
call dtgex2 ( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, n1,
    n2, work, lwork, info )
call ctgex2 ( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1,  info )
call ztgex2 ( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, info )
```

Description

The real routines `stgex2/dtgex2` swap adjacent diagonal blocks ($A11$, $B11$) and ($A22$, $B22$) of size 1-by-1 or 2-by-2 in an upper (quasi) triangular matrix pair *(A, B)* by an orthogonal equivalence transformation. (*A, B*) must be in generalized real Schur canonical form (as returned by `sgges/dgges`), that is, *A* is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks. *B* is upper triangular.

The complex routines `ctgex2/ztgex2` swap adjacent diagonal 1-by-1 blocks ($A11$, $B11$) and ($A22$, $B22$) in an upper triangular matrix pair *(A, B)* by an unitary equivalence transformation. (*A, B*) must be in generalized Schur canonical form, that is, *A* and *B* are both upper triangular.

All routines optionally update the matrices *Q* and *Z* of generalized Schur vectors:

$Q(\text{in}) * A(\text{in}) * Z(\text{in})' = Q(\text{out}) * A(\text{out}) * Z(\text{out})'$
$Q(\text{in}) * B(\text{in}) * Z(\text{in})' = Q(\text{out}) * B(\text{out}) * Z(\text{out})'$

Input Parameters

wantq — `LOGICAL`.
If *wantq* = `.TRUE.` : update the left transformation matrix *Q;*
If *wantq* = `.FALSE.`: do not update *Q*.

wantz — `LOGICAL`.
If *wantz* = `.TRUE.` : update the right transformation matrix *Z;*
If *wantz* = `.FALSE.`: do not update *Z*.

n	INTEGER. The order of the matrices *A* and *B*. *n* ≥ 0.
a, b	REAL for stgex2 DOUBLE PRECISION for dtgex2 COMPLEX for ctgex2 COMPLEX*16 for ztgex2. Arrays, DIMENSION (*lda*, *n*) and (*ldb*, *n*), respectively. On entry, the matrices *A* and *B* in the pair (*A*, *B*).
lda	INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*n*).
ldb	INTEGER. The leading dimension of the array *b*. *ldb* ≥ max(1,*n*).
q, z	REAL for stgex2 DOUBLE PRECISION for dtgex2 COMPLEX for ctgex2 COMPLEX*16 for ztgex2. Arrays, DIMENSION (*ldq*, *n*) and (*ldz*, *n*), respectively. On entry, if *wantq* = .TRUE., *q* contains the orthogonal/unitary matrix *Q,* and if *wantz* = .TRUE., *z* contains the orthogonal/unitary matrix *Z*.
ldq	INTEGER. The leading dimension of the array *q*. *ldq* ≥ 1. If *wantq* = .TRUE., *ldq* ≥ *n*.
ldz	INTEGER. The leading dimension of the array *z*. *ldz* ≥ 1. If *wantz* = .TRUE., *ldz* ≥ *n*.
j1	INTEGER. The index to the first block (*A*11, *B*11). 1 ≤ *j1* ≤ *n*.
n1	INTEGER. Used with real flavors only. The order of the first block (*A*11, *B*11). *n1* = 0, 1 or 2.
n2	INTEGER. Used with real flavors only. The order of the second block (*A*22, *B*22). *n2* = 0, 1 or 2.
work	REAL for stgex2 DOUBLE PRECISION for dtgex2. Workspace array, DIMENSION (*lwork*). Used with real flavors only.

lwork INTEGER.
The dimension of the array *work*.
lwork ≥ max(*n**(*n2*+*n1*), 2*(*n2*+*n1*)^2)

Output Parameters

a On exit, the updated matrix *A*.

b On exit, the updated matrix *B*.

q On exit, the updated matrix *Q*.
Not referenced if *wantq* = .FALSE..

z On exit, the updated matrix *Z*.
Not referenced if *wantz* = .FALSE..

info INTEGER.
=0: Successful exit
For stgex2/dtgex2: if *info* = 1, the transformed matrix (*A*, *B*) would be too far from generalized Schur form; the blocks are not swapped and (*A*, *B*) and (*Q*, *Z*) are unchanged. The problem of swapping is too ill-conditioned. If *info* = -16: *lwork* is too small. Appropriate value for *lwork* is returned in *work*(1).

For ctgex2/ztgex2: if *info* = 1, the transformed matrix pair (*A*, *B*) would be too far from generalized Schur form; the problem is ill-conditioned. (*A, B*) may have been partially reordered, and *ilst* points to the first row of the current position of the block being moved.

?tgsy2

Solves the generalized Sylvester equation (unblocked algorithm).

Syntax

```
call stgsy2 ( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde,
    f, ldf, scale, rdsum, rdscal, iwork, pq, info )
call dtgsy2 ( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde,
    f, ldf, scale, rdsum, rdscal, iwork, pq, info )
call ctgsy2 ( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde,
    f, ldf, scale, rdsum, rdscal, iwork, pq, info )
```

```
call ztgsy2 ( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde,
    f, ldf, scale, rdsum, rdscal, iwork, pq, info )
```

Description

The routine ?tgsy2 solves the generalized Sylvester equation:

$$AR - LB = \text{scale}*C \qquad (1)$$
$$DR - LE = \text{scale}*F,$$

using Level 1 and 2 BLAS, where R and L are unknown m-by-n matrices, (A, D), (B, E) and (C, F) are given matrix pairs of size m-by-m, n-by-n and m-by-n, respectively.
For stgsy2/dtgsy2, pairs (A, D) and (B, E) must be in generalized Schur canonical form, that is, A, B are upper quasi triangular and D, E are upper triangular. For ctgsy2/ztgsy2, matrices A, B, D and E are upper triangular (that is, (A, D) and (B, E) in generalized Schur form).

The solution (R, L) overwrites (C, F). $0 \le scale \le 1$ is an output scaling factor chosen to avoid overflow.

In matrix notation, solving equation (1) corresponds to solve

$$Zx = \text{scale}*b,$$

where Z is defined as

$$Z = \begin{bmatrix} \text{kron}(I_n, A) & -\text{kron}(B', I_m) \\ \text{kron}(I_n, D) & -\text{kron}(E', I_m) \end{bmatrix} \qquad (2)$$

Here I_k is the identity matrix of size k and X' is the transpose of X.
kron(X, Y) denotes the Kronecker product between the matrices X and Y.

If *trans* = 'T', solve the transposed (conjugate transposed) system

$$Z'y = \text{scale}*b$$

for y, which is equivalent to solve for R and L in

$$A'R + D'L = \text{scale}*C \qquad (3)$$
$$RB' + LE' = \text{scale}*(-F)$$

This case is used to compute an estimate of Dif[(A, D), (B, E)] = $sigma\_min(Z)$ using reverse communication with ?lacon.

?tgsy2 also (for $ijob \ge 1$) contributes to the computation in ?tgsyl of an upper bound on the separation between two matrix pairs. Then the input (A, D), (B, E) are sub-pencils of the matrix pair (two matrix pairs) in ?tgsyl. See ?tgsyl for details.

Input Parameters

trans CHARACTER
If *trans* = 'N', solve the generalized Sylvester equation (1);
If *trans* = 'T': solve the 'transposed' system (3).

ijob INTEGER.
Specifies what kind of functionality is to be performed.
If *ijob* = 0: solve (1) only.
If *ijob* = 1: a contribution from this subsystem to a Frobenius norm-based estimate of the separation between two matrix pairs is computed (look ahead strategy is used);
If *ijob* = 2: a contribution from this subsystem to a Frobenius norm-based estimate of the separation between two matrix pairs is computed (?gecon on sub-systems is used).
Not referenced if *trans* = 'T'.

m INTEGER.
On entry, *m* specifies the order of *A* and *D*, and the row dimension of *C*, *F*, *R* and *L*.

n INTEGER.
On entry, *n* specifies the order of *B* and *E*, and the column dimension of *C*, *F*, *R* and *L*.

a, b REAL for stgsy2
DOUBLE PRECISION for dtgsy2
COMPLEX for ctgsy2
COMPLEX*16 for ztgsy2.
Arrays, DIMENSION (*lda*, *m*) and (*ldb, n*), respectively. On entry, *a* contains an upper (quasi) triangular matrix *A* and *b* contains an upper (quasi) triangular matrix *B*.

lda INTEGER.
The leading dimension of the array *a*. *lda* ≥ max(1, *m*).

ldb INTEGER.
The leading dimension of the array *b*. *ldb* ≥ max(1, *n*).

c, f REAL for stgsy2
DOUBLE PRECISION for dtgsy2
COMPLEX for ctgsy2
COMPLEX*16 for ztgsy2.

Arrays, DIMENSION (*ldc, n*) and (*ldf, n*), respectively. On entry, *c* contains the right-hand-side of the first matrix equation in (1) and *f* contains the right-hand-side of the second matrix equation in (1).

ldc — INTEGER.
The leading dimension of the array *c*. *ldc* ≥ max(1, *m*).

d, e — REAL for stgsy2
DOUBLE PRECISION for dtgsy2
COMPLEX for ctgsy2
COMPLEX*16 for ztgsy2.
Arrays, DIMENSION (*ldd*, *m*) and (*lde, n*), respectively. On entry, *d* contains an upper triangular matrix *D* and *e* contains an upper triangular matrix *E*.

ldd — INTEGER.
The leading dimension of the array *d*. *ldd* ≥ max(1, *m*).

lde — INTEGER.
The leading dimension of the array *e*. *lde* ≥ max(1, *n*).

ldf — INTEGER.
The leading dimension of the array *f*. *ldf* ≥ max(1, *m*).

rdsum — REAL for stgsy2/ctgsy2
DOUBLE PRECISION for dtgsy2/ztgsy2.
On entry, the sum of squares of computed contributions to the Dif-estimate under computation by ?tgsyl, where the scaling factor *rdscal* has been factored out.

rdscal — REAL for stgsy2/ctgsy2
DOUBLE PRECISION for dtgsy2/ztgsy2.
On entry, scaling factor used to prevent overflow in *rdsum*.

iwork — INTEGER. Used with real flavors only.
Workspace array, DIMENSION (*m*+*n*+2).

Output Parameters

c — On exit, if *ijob* = 0, *c* has been overwritten by the solution *R*.

f — On exit, if *ijob* = 0, *f* has been overwritten by the solution *L*.

scale — REAL for stgsy2/ctgsy2
DOUBLE PRECISION for dtgsy2/ztgsy2.
On exit, 0 ≤ *scale* ≤ 1. If 0 < *scale* < 1, the solutions *R* and *L* (*C* and *F* on

entry) will hold the solutions to a slightly perturbed system, but the input matrices A, B, D and E have not been changed. If `scale` = 0, R and L will hold the solutions to the homogeneous system with $C = F = 0$. Normally `scale` = 1.

rdsum — On exit, the corresponding sum of squares updated with the contributions from the current sub-system.
If *trans* = 'T', *rdsum* is not touched.
Note that *rdsum* only makes sense when `?tgsy2` is called by `?tgsyl`.

rdscal — On exit, *rdscal* is updated with respect to the current contributions in *rdsum*.
If *trans* = 'T', *rdscal* is not touched.
Note that *rdscal* only makes sense when `?tgsy2` is called by `?tgsyl`.

pq — `INTEGER`. Used with real flavors only.
On exit, the number of subsystems (of size 2-by-2, 4-by-4 and 8-by-8) solved by the routine `stgsy2/dtgsy2`.

info — `INTEGER`.
On exit, if *info* is set to
=0: Successful exit
<0: If *info* = *-i*, the *i*-th argument had an illegal value.
>0: The matrix pairs (A, D) and (B, E) have common or very close eigenvalues.

?trti2

Computes the inverse of a triangular matrix (unblocked algorithm).

Syntax

```
call strti2 ( uplo, diag, n, a, lda, info )
call dtrti2 ( uplo, diag, n, a, lda, info )
call ctrti2 ( uplo, diag, n, a, lda, info )
call ztrti2 ( uplo, diag, n, a, lda, info )
```

Description

The routine `?trti2` computes the inverse of a real/complex upper or lower triangular matrix.

This is the Level 2 BLAS version of the algorithm.

Input Parameters

uplo CHARACTER*1.
Specifies whether the matrix *A* is upper or lower triangular.
= 'U': Upper triangular
= 'L': Lower triangular

diag CHARACTER*1.
Specifies whether or not the matrix *A* is unit triangular.
= 'N': Non-unit triangular
= 'U': Unit triangular

n INTEGER.
The order of the matrix *A*. $n \geq 0$.

a REAL for strti2
DOUBLE PRECISION for dtrti2
COMPLEX for ctrti2
COMPLEX*16 for ztrti2.
Array, DIMENSION (*lda*, *n*).
On entry, the triangular matrix *A*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of the array *a* contains the upper triangular matrix, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the array *a* contains the lower triangular matrix, and the strictly upper triangular part of *a* is not referenced. If *diag* = 'U', the diagonal elements of *a* are also not referenced and are assumed to be 1.

lda INTEGER.
The leading dimension of the array *a*. $lda \geq \max(1,n)$.

Output Parameters

a On exit, the (triangular) inverse of the original matrix, in the same storage format.

info INTEGER.
= 0: successful exit
< 0: if *info* = -*k*, the *k*-th argument had an illegal value

Utility Functions and Routines

This section describes LAPACK utility functions and routines. Summary information about these routines is given in the following table:

Table 5-2 LAPACK Utility Routines

Routine Name	Data Types	Description
`ilaenv`		Environmental enquiry function which returns values for tuning algorithmic performance.
`ieeeck`		Checks if the infinity and NaN arithmetic is safe. Called by `ilaenv`.
`lsame`		Tests two characters for equality regardless of case.
`lsamen`		Tests two character strings for equality regardless of case.
`?labad`	s,d	Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.
`?lamch`	s,d	Determines machine parameters for floating-point arithmetic.
`?lamc1`	s,d	Called from `?lamc2`. Determines machine parameters given by *beta, t, rnd, ieee1*.
`?lamc2`	s,d	Used by `?lamch`. Determines machine parameters specified in its arguments list.
`?lamc3`	s,d	Called from `?lamc1-?lamc5`. Intended to force *a* and *b* to be stored prior to doing the addition of *a* and *b*.
`?lamc4`	s,d	This is a service routine for `?lamc2`.
`?lamc5`	s,d	Called from `?lamc2`. Attempts to compute the largest machine floating-point number, without overflow.
`second/ dsecnd`		Return user time for a process.
`xerbla`		Error handling routine called by LAPACK routines.

ilaenv

Environmental enquiry function which returns values for tuning algorithmic performance.

Syntax

```
value = ilaenv ( ispec, name, opts, n1, n2, n3, n4 )
```

Description

Enquiry function `ilaenv` is called from the LAPACK routines to choose problem-dependent parameters for the local environment. See *`ispec`* for a description of the parameters.

This version provides a set of parameters which should give good, but not optimal, performance on many of the currently available computers. Users are encouraged to modify this subroutine to set the tuning parameters for their particular machine using the option and problem size information in the arguments.

This routine will not function correctly if it is converted to all lower case. Converting it to all upper case is allowed.

Input Parameters

`ispec` `INTEGER`. Specifies the parameter to be returned as the value of `ilaenv`:

= 1: the optimal blocksize; if this value is 1, an unblocked algorithm will give the best performance.

= 2: the minimum block size for which the block routine should be used; if the usable block size is less than this value, an unblocked routine should be used.

= 3: the crossover point (in a block routine, for *N* less than this value, an unblocked routine should be used)

= 4: the number of shifts, used in the nonsymmetric eigenvalue routines

= 5: the minimum column dimension for blocking to be used; rectangular blocks must have dimension at least `k` by `m`, where `k` is given by `ilaenv`(2,...) and `m` by `ilaenv`(5,...)

= 6: the crossover point for the SVD (when reducing an `m` by `n` matrix to bidiagonal form, if max(`m,n`)/min(`m,n`) exceeds this value, a *QR* factorization is used first to reduce the matrix to a triangular form.)

= 7: the number of processors

= 8: the crossover point for the multishift *QR* and *QZ* methods for nonsymmetric eigenvalue problems.

= 9: maximum size of the subproblems at the bottom of the computation tree in the divide-and-conquer algorithm (used by `?gelsd` and `?gesdd`)

=10: IEEE NaN arithmetic can be trusted not to trap

=11: infinity arithmetic can be trusted not to trap *name* — `CHARACTER*(*)`.The name of the calling subroutine, in either upper case or lower case.

opts — `CHARACTER*(*)`. The character options to the subroutine *name*, concatenated into a single character string. For example, *uplo* = `'U'`, *trans* = `'T'`, and *diag* = `'N'` for a triangular routine would be specified as *opts* = `'UTN'`.

n1,n2,n3,n4 — `INTEGER`. Problem dimensions for the subroutine *name*; these may not all be required.

Output Parameters

value — `INTEGER`.
If *value* ≥ 0: the value of the parameter specified by *ispec*;
If *value* = -k < 0: the k-th argument had an illegal value.

Application Notes

The following conventions have been used when calling `ilaenv` from the LAPACK routines:

1) *opts* is a concatenation of all of the character options to subroutine *name*, in the same order that they appear in the argument list for *name*, even if they are not used in determining the value of the parameter specified by *ispec*.

2) The problem dimensions *n1,n2,n3,n4* are specified in the order that they appear in the argument list for *name*. *n1* is used first, *n2* second, and so on, and unused problem dimensions are passed a value of -1.

3) The parameter value returned by `ilaenv` is checked for validity in the calling subroutine. For example, `ilaenv` is used to retrieve the optimal blocksize for `strtri` as follows:

```
nb = ilaenv( 1, 'strtri', uplo // diag, n, -1, -1, -1 )
if( nb.le.1 ) nb = max( 1, n )
```

ieeeck

Checks if the infinity and NaN arithmetic is safe. Called by `ilaenv`.

Syntax

`ival = ieeeck( ispec, zero, one )`

Description

The function `ieeeck` is called from the `ilaenv` to verify that infinity and possibly `NaN` arithmetic is safe, that is, will not trap.

Input Parameters

ispec — `INTEGER`. Specifies whether to test just for inifinity arithmetic or both for infinity and `NaN` arithmetic:
If *ispec* = 0: Verify infinity arithmetic only.
If *ispec* = 1: Verify infinity and `NaN` arithmetic.

zero — `REAL`. Must contain the value 0.0
This is passed to prevent the compiler from optimizing away this code.

one — `REAL`. Must contain the value 1.0
This is passed to prevent the compiler from optimizing away this code.

Output Value

ival — `INTEGER`.
If *ival* = 0: Arithmetic failed to produce the correct answers.
If *ival* = 1: Arithmetic produced the correct answers.

lsame

Tests two characters for equality regardless of case.

Syntax

`val = lsame ( ca, cb )`

Description

This logical function returns `.TRUE.` if *ca* is the same letter as *cb* regardless of case.

Input Parameters

ca, cb	`CHARACTER*1`. Specify the single characters to be compared.

Output Parameters

val	`LOGICAL`. Result of the comparison.

lsamen

Tests two character strings for equality regardless of case.

Syntax

`val = lsamen ( n, ca, cb )`

Description

This logical function tests if the first *n* letters of the string *ca* are the same as the first *n* letters of *cb*, regardless of case. The function `lsamen` returns `.TRUE.` if *ca* and *cb* are equivalent except for case and `.FALSE.` otherwise. `lsamen` also returns `.FALSE.` if `len(ca)` or `len(cb)` is less than *n*.

Input Parameters

n	`INTEGER`. The number of characters in *ca* and *cb* to be compared.

ca, cb — `CHARACTER*(*)`. Specify two character strings of length at least *n* to be compared. Only the first *n* characters of each string will be accessed.

Output Parameters

val — `LOGICAL`. Result of the comparison.

?labad

Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.

Syntax

```
call slabad ( small, large )
call dlabad ( small, large )
```

Description

This routine takes as input the values computed by `slamch/dlamch` for underflow and overflow, and returns the square root of each of these values if the log of *large* is sufficiently large. This subroutine is intended to identify machines with a large exponent range, such as the Crays, and redefine the underflow and overflow limits to be the square roots of the values computed by `?lamch`. This subroutine is needed because `?lamch` does not compensate for poor arithmetic in the upper half of the exponent range, as is found on a Cray.

Input Parameters

small — `REAL` for `slabad`
`DOUBLE PRECISION` for `dlabad`.
The underflow threshold as computed by `?lamch`.

large — `REAL` for `slabad`
`DOUBLE PRECISION` for `dlabad`.
The overflow threshold as computed by `?lamch`.

Output Parameters

small — On exit, if log10(*large*) is sufficiently large, the square root of *small*, otherwise unchanged.

large On exit, if log10(*large*) is sufficiently large, the square root of *large*, otherwise unchanged.

?lamch

Determines machine parameters for floating-point arithmetic.

Syntax

```
val = slamch ( cmach )
val = dlamch ( cmach )
```

Description

The function `?lamch` determines single precision and double precision machine parameters.

Input Parameters

cmach `CHARACTER*1`. Specifies the value to be returned by `?lamch`:
= '*E*' or '*e*', *val* = *eps*
= '*S*' or '*s* , *val* = *sfmin*
= '*B*' or '*b*', *val* = *base*
= '*P*' or '*p*', *val* = *eps*base*
= '*N*' or '*n*', *val* = *t*
= '*R*' or '*r*', *val* = *rnd*
= '*M*' or '*m*', *val* = *emin*
= '*U*' or '*u*', *val* = *rmin*
= '*L*' or '*l*', *val* = *emax*
= '*O*' or '*o*', *val* = *rmax*
where
eps = relative machine precision;
sfmin = safe minimum, such that 1/*sfmin* does not overflow;
base = base of the machine;
prec = *eps*base*;
t = number of (base) digits in the mantissa;
rnd = 1.0 when rounding occurs in addition, 0.0 otherwise;
emin = minimum exponent before (gradual) underflow;

rmin = *underflow_threshold* - *base***(*emin*-1);
emax = largest exponent before overflow;
rmax = *overflow_threshold* - (*base****emax*)*(1-*eps*).

Output Parameters

val — `REAL` for `slamch`
`DOUBLE PRECISION` for `dlamch`
Value returned by the function.

?lamc1

Called from `?lamc2`.
Determines machine parameters given by *beta, t, rnd, ieee1*.

Syntax

```
call slamc1 ( beta, t, rnd, ieee1 )
call dlamc1 ( beta, t, rnd, ieee1 )
```

Description

The routine `?lamc1` determines machine parameters given by *beta, t, rnd, ieee1*.

Output Parameters

beta — `INTEGER`. The base of the machine.

t — `INTEGER`. The number of (*beta*) digits in the mantissa.

rnd — `LOGICAL`.
Specifies whether proper rounding (*rnd* = `.TRUE.`) or chopping (*rnd* = `.FALSE.`) occurs in addition. This may not be a reliable guide to the way in which the machine performs its arithmetic.

ieee1 — `LOGICAL`.
Specifies whether rounding appears to be done in the *ieee* 'round to nearest' style.

?lamc2

Used by `?lamch`.
Determines machine parameters specified in its arguments list.

Syntax

```
call slamc2 ( beta, t, rnd, eps, emin, rmin, emax, rmax )
call dlamc2 ( beta, t, rnd, eps, emin, rmin, emax, rmax )
```

Description

The routine `?lamc2` determines machine parameters specified in its arguments list.

Output Parameters

beta — `INTEGER`. The base of the machine.

t — `INTEGER`. The number of (*beta*) digits in the mantissa.

rnd — `LOGICAL`.
Specifies whether proper rounding (*rnd* = `.TRUE.`) or chopping (*rnd* = `.FALSE.`) occurs in addition. This may not be a reliable guide to the way in which the machine performs its arithmetic.

eps — `REAL` for `slamc2`
`DOUBLE PRECISION` for `dlamc2`
The smallest positive number such that
$fl(1.0 - eps) < 1.0$,
where *fl* denotes the computed value.

emin — `INTEGER`. The minimum exponent before (gradual) underflow occurs.

rmin — `REAL` for `slamc2`
`DOUBLE PRECISION` for `dlamc2`
The smallest normalized number for the machine, given by $base^{emin - 1}$, where *base* is the floating point value of *beta*.

emax — `INTEGER`.The maximum exponent before overflow occurs.

rmax REAL for slamc2
DOUBLE PRECISION for dlamc2
The largest positive number for the machine, given by $base^{emax}(1 - eps)$, where *base* is the floating point value of *beta*.

?lamc3

Called from ?lamc1-?lamc5. *Intended to force* a *and* b *to be stored prior to doing the addition of* a *and* b.

Syntax

```
val = slamc3 (a, b)
val = dlamc3 (a, b)
```

Description

The routine is intended to force *a* and *b* to be stored prior to doing the addition of *a* and *b*, for use in situations where optimizers might hold one of these in a register.

Input Parameters

a,*b* REAL for slamc3
DOUBLE PRECISION for dlamc3
The values *a* and *b*.

Output Parameters

val REAL for slamc3
DOUBLE PRECISION for dlamc3
The result of adding values *a* and *b*.

?lamc4

This is a service routine for `?lamc2`.

Syntax

```
call slamc4 (emin, start, base)
call dlamc4 (emin, start, base)
```

Description

This is a service routine for `?lamc2.`

Input Parameters

start	`REAL` for `slamc4` `DOUBLE PRECISION` for `dlamc4` The starting point for determining *emin*.
base	`INTEGER`. The base of the machine.

Output Parameters

emin	`INTEGER`. The minimum exponent before (gradual) underflow, computed by setting *a* = *start* and dividing by *base* until the previous *a* can not be recovered.

?lamc5

Called from `?lamc2`.
Attempts to compute the largest machine floating-point number, without overflow.

Syntax

```
call slamc5 ( beta, p, emin, ieee, emax, rmax )
call dlamc5 ( beta, p, emin, ieee, emax, rmax )
```

Description

The routine `?lamc5` attempts to compute *rmax*, the largest machine floating-point number, without overflow. It assumes that *emax* + abs(*emin*) sum approximately to a power of 2. It will fail on machines where this assumption does not hold, for example, the Cyber 205 (*emin* = -28625, *emax* = 28718). It will also fail if the value supplied for *emin* is too large (that is, too close to zero), probably with overflow.

Input Parameters

beta	`INTEGER.` The base of floating-point arithmetic.
p	`INTEGER.` The number of base *beta* digits in the mantissa of a floating-point value.
emin	`INTEGER.` The minimum exponent before (gradual) underflow.
ieee	`LOGICAL.` A logical flag specifying whether or not the arithmetic system is thought to comply with the IEEE standard.

Output Parameters.

emax	`INTEGER.` The largest exponent before overflow.
rmax	`REAL` for `slamc5` `DOUBLE PRECISION` for `dlamc5` The largest machine floating-point number.

second/dsecnd

Return user time for a process.

Syntax

```
val = second()
val = dsecnd()
```

Description

The functions `second/dsecnd` return the user time for a process in seconds. These versions get the time from the system function `etime`. The difference is that `dsecnd` returns the result with double presision.

Output Parameters

val `REAL` for `second`
`DOUBLE PRECISION` for `dsecnd`
User time for a process.

xerbla

Error handling routine called by LAPACK routines.

Syntax

```
call xerbla ( srname, info )
```

Description

The routine `xerbla` is an error handler for the LAPACK routines. It is called by a LAPACK routine if an input parameter has an invalid value.
A message is printed and execution stops.
Installers may consider modifying the `stop` statement in order to call system-specific exception-handling facilities.

Input Parameters

srname `CHARACTER*6`
The name of the routine which called `xerbla`.

info `INTEGER`.
The position of the invalid parameter in the parameter list of the calling routine.

ScaLAPACK Routines 6

This chapter describes the Intel® Math Kernel Library implementation of routines from the ScaLAPACK package for distributed-memory architectures.Routines are supported for both real and complex dense and band matrices to perform the tasks of solving systems of linear equations, solving linear least-squares problems, eigenvalue and singular value problems, as well as performing a number of related computational tasks. All routines are available in both single precision and double precision.

NOTE. ScaLAPACK routines are provided with Intel® Cluster MKL product only which is a superset of Intel MKL.

Sections in this chapter include descriptions of ScaLAPACK computational routines that perform distinct computational tasks, as well as driver routines for solving standard types of problems in one call.

Generally, ScaLAPACK runs on a network of computers using MPI as a message-passing layer and a set of prebuilt communication subprograms (BLACS), as well as a set of BLAS optimized for the target architecture. Intel Cluster MKL version of ScaLAPACK is optimized for Intel® processors and uses MPICH version of MPI. For the detailed system requirements, see *Intel MKL Release Notes* and *Intel MKL Technical UserNotes*.

For full reference on ScaLAPACK routines and related information see [SLUG].

Overview

The model of the computing environment for ScaLAPACK is represented as a one-dimensional array of processes (for operations on band or tridiagonal matrices) or also a two-dimensional process grid (for operations on dense matrices). To use ScaLAPACK, all global matrices or vectors should be distributed on this array or grid prior to calling the ScaLAPACK routines.

ScaLAPACK uses the two-dimensional block-cyclic data distribution as a layout for dense matrix computations.
This distribution provides good work balance between available processors, as well as gives the opportunity to use BLAS Level 3 routines for optimal local computations. Information about the data distribution that is required to establish the mapping between each global array and its corresponding process and memory location is contained in the so called *array descriptor* associated with each global array.
An example of an array descriptor structure is given in Table 6-1

Table 6-1 Content of the array descriptor for dense matrices

Array Element #	Name	Definition
1	`dtype`	Descriptor type (=1 for dense matrices)
2	`ctxt`	BLACS context handle for the process grid
3	`m`	Number of rows in the global array
4	`n`	Number of columns in the global array
5	`mb`	Row blocking factor
6	`nb`	Column blocking factor
7	`rsrc`	Process row over which the first row of the global array is distributed
8	`csrc`	Process column over which the first column of the global array is distributed
9	`lld`	Leading dimension of the local array

The number of rows and columns of a global dense matrix that a particular process in a grid receives after data distributing is denoted by $LOC_r()$ and $LOC_c()$, respectively. To compute these numbers, you can use the ScaLAPACK tool routine `numroc`.

After the block-cyclic distribution of global data is done, you may choose to perform an operation on a submatrix of the global matrix A, which is contained in the global subarray sub(A), defined by the following 6 values (for dense matrices):

m The number of rows of sub(A)

n	The number of columns of sub(*A*)
a	A pointer to the local array containing the entire global array *A*
ia	The row index of sub(*A*) in the global array
ja	The column index of sub(*A*) in the global array
desca	The array descriptor for the global array

Routine Naming Conventions

For each routine introduced in this chapter, you can use the ScaLAPACK name. The naming convention for ScaLAPACK routines is similar to that used for LAPACK routines (see page 4-2 in Chapter 4 and page 5-4 in Chapter 5). A general rule is that each routine name in ScaLAPACK, which has an LAPACK equivalent, is simply the LAPACK name prefixed by initial letter p.

ScaLAPACK names have the structure **pxyyzzz** or **pxyyzz**, which is described below.

The initial letter **p** is a distinctive prefix of ScaLAPACK routines and is present in each such routine.
The second letter **x** indicates the data type:

s	real, single precision	c	complex, single precision
d	real, double precision	z	complex, double precision

The second and third letters **yy** indicate the matrix type as:

ge	general
gb	general band
gg	a pair of general matrices (for a generalized problem)
dt	general tridiagonal (diagonally dominant-like)
db	general band (diagonally dominant-like)
po	symmetric or Hermitian positive-definite
pb	symmetric or Hermitian positive-definite band
pt	symmetric or Hermitian positive-definite tridiagonal
sy	symmetric
st	symmetric tridiagonal (real)
he	Hermitian
or	orthogonal
tr	triangular (or quasi-triangular)
tz	trapezoidal
un	unitary

For computational routines, the last three letters **zzz** indicate the computation performed and have the same meaning as for LAPACK routines.

For driver routines, the last two letters **zz** or three letters **zzz** have the following meaning:

`sv` a *simple* driver for solving a linear system
`svx` an *expert* driver for solving a linear system
`ls` a driver for solving a linear least squares problem
`ev` a simple driver for solving a symmetric eigenvalue problem
`evx` an expert driver for solving a symmetric eigenvalue problem
`svd` a driver for computing a singular value decomposition
`gvx` an expert driver for solving a generalized symmetric definite eigenvalue problem

Simple driver here mean that the driver just solves the general problem, whereas an *expert* driver is more versatile and can also optionally perform some related computations (such, for example, as refining the solution and computing error bounds after the linear system is solved).

Computational Routines

In the sections that follow, the descriptions of ScaLAPACK computational routines are given. These routines perform distinct computational tasks that can be used for:

- Solving Systems of Linear Equations
- Orthogonal Factorizations and LLS Problems
- Symmetric Eigenproblems
- Nonsymmetric Eigenvalue Problems
- Singular Value Decomposition
- Generalized Symmetric-Definite Eigenproblems

See also the respective driver routines.

Linear Equations

ScaLAPACK supports routines for the systems of equations with the following types of matrices:

- general
- general banded
- general diagonally dominant-like banded (including general tridiagonal)
- symmetric or Hermitian positive-definite
- symmetric or Hermitian positive-definite banded
- symmetric or Hermitian positive-definite tridiagonal

A *diagonally dominant-like* matrix is defined as a matrix for which it is known in advance that pivoting is not required in the LU factorization of this matrix.

For the above matrix types, the library includes routines for performing the following computations: *factoring* the matrix; *equilibrating* the matrix; *solving* a system of linear equations; *estimating the condition number* of a matrix; *refining* the solution of linear equations and computing its error bounds; *inverting* the matrix. Note that for some of the listed matrix types only part of the computational routines are provided (for example, routines that refine the solution are not provided for band or tridiagonal matrices). See Table 6-2 for full list of available routines.

To solve a particular problem, you can either call two or more computational routines or call a corresponding driver routine that combines several tasks in one call. Thus, to solve a system of linear equations with a general matrix, you can first call `p?getrf` (*LU* factorization) and then `p?getrs` (computing the solution). Then, you might wish to call `p?gerfs` to refine the solution and get the error bounds. Alternatively, you can just use the driver routine `p?gesvx` which performs all these tasks in one call.

Table 6-2 lists the ScaLAPACK computational routines for factorizing, equilibrating, and inverting matrices, estimating their condition numbers, solving systems of equations with real matrices, refining the solution, and estimating its error.

Table 6-2 Computational Routines for Systems of Linear Equations

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general (partial pivoting)	p?getrf	p?geequ	p?getrs	p?gecon	p?gerfs	p?getri
general band (partial pivoting)	p?gbtrf		p?gbtrs			
general band (no pivoting)	p?dbtrf		p?dbtrs			
general tridiagonal (no pivoting)	p?dttrf		p?dttrs			
symmetric/Hermitian positive-definite	p?potrf	p?poequ	p?potrs	p?pocon	p?porfs	p?potri
symmetric/Hermitian positive-definite, band	p?pbtrf		p?pbtrs			
symmetric/Hermitian positive-definite, tridiagonal	p?pttrf		p?pttrs			
triangular			p?trtrs	p?trcon	p?trrfs	p?trtri

In this table **?** stands for **s** (single precision real), **d** (double precision real), **c** (single precision complex), or **z** (double precision complex).

Routines for Matrix Factorization

This section describes the ScaLAPACK routines for matrix factorization. The following factorizations are supported:

- LU factorization of general matrices
- LU factorization of diagonally dominant-like matrices
- Cholesky factorization of real symmetric or complex Hermitian positive-definite matrices

You can compute the factorizations using full and band storage of matrices.

p?getrf

Computes the LU factorization of a general m by n distributed matrix.

Syntax

```
call psgetrf ( m, n, a, ia, ja, desca, ipiv, info )
call pdgetrf ( m, n, a, ia, ja, desca, ipiv, info )
call pcgetrf ( m, n, a, ia, ja, desca, ipiv, info )
call pzgetrf ( m, n, a, ia, ja, desca, ipiv, info )
```

Description

The routine forms the LU factorization of a general m-by-n distributed matrix sub(A) = A(*ia:ia+n-1*, *ja:ja+n-1*) as

$$A = P\,L\,U$$

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$). L and U are stored in sub(A).

The routine uses partial pivoting, with row interchanges.

Input Parameters

m — (global) INTEGER. The number of rows in the distributed submatrix sub(*A*); *m* ≥ 0.

n — (global) INTEGER. The number of columns in the distributed submatrix sub(*A*); *n* ≥ 0.

a — (local)

REAL for psgetrf
DOUBLE PRECISION for pdgetrf
COMPLEX for pcgetrf
DOUBLE COMPLEX for pzgetrf.
Pointer into the local memory to an array of local dimension (*lld_a*, LOC_c(*ja*+*n*-1)).
Contains the local pieces of the distributed matrix sub(*A*) to be factored.

ia, ja — (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1), respectively.

desca — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

a — Overwritten by local pieces of the factors *L* and *U* from the factorization *A* = *P L U*. The unit diagonal elements of *L* are not stored.

ipiv — (local) INTEGER array.
The dimension of *ipiv* is (LOC_r(*m_a*)+ *mb_a*).
This array contains the pivoting information: local row *i* was interchanged with global row *ipiv*(*i*). This array is tied to the distributed matrix *A*.

info — (global) INTEGER.

If *info*=0, the execution is successful.
info < 0: if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.
If *info* = *i*, u_{ii} is 0. The factorization has been completed, but the factor *U* is exactly singular. Division by zero will occur if you use the factor *U* for solving a system of linear equations.

p?gbtrf

Computes the LU factorization of a general n-by-n banded distributed matrix.

Syntax

```
call psgbtrf ( n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork,
    info )
call pdgbtrf ( n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork,
    info )
call pcgbtrf ( n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork,
    info )
call pzgbtrf ( n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork,
    info )
```

Description

The routine computes the *LU* factorization of a general *n*-by-*n* real/complex banded distributed matrix A(*1:n*, *ja:ja+n-1*) using partial pivoting with row interchanges.

The resulting factorization is not the same factorization as returned from the LAPACK routine `?gbtrf`. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form

$$A(1:n, ja:ja+n-1) = P\ L\ U\ Q$$

where P and Q are permutation matrices, and L and U are banded lower and upper triangular matrices, respectively. The matrix Q represents reordering of columns for the sake of parallelism, while P represents reordering of rows for numerical stability using classic partial pivoting.

Input Parameters

n	(global) `INTEGER`. The number of rows and columns in the distributed submatrix A(*1:n*, *ja:ja+n-1*); $n \geq 0$.
bwl	(global) `INTEGER`. The number of sub-diagonals within the band of A ($0 \leq bwl \leq n-1$).
bwu	(global) `INTEGER`. The number of super-diagonals within the band of A ($0 \leq bwu \leq n-1$).

a (local)

REAL for psgbtrf
DOUBLE PRECISION for pdgbtrf
COMPLEX for pcgbtrf
DOUBLE COMPLEX for pzgbtrf.
Pointer into the local memory to an array of local dimension (*lld_a*, LOC_c(*ja*+*n*-*1*)) where
lld_a ≥ 2**bwl* + 2**bwu* +*1*.
Contains the local pieces of the *n*-by-*n* distributed banded matrix
A(*1*:*n*, *ja*:*ja*+*n*-*1*) to be factored.

ja (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7;
else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

laf (local) INTEGER. The dimension of the array *af*.
Must be *laf* ≥ (NB+*bwu*)*(*bwl*+*bwu*)+6*(*bwl*+*bwu*)*(*bwl*+2**bwu*) .

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work (local) Same type as *a*. Workspace array of dimension *lwork* .

lwork (local or global) INTEGER. The size of the *work* array (*lwork* ≥ 1). If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

Output Parameters

a On exit, this array contains details of the factorization. Note that additional permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.

ipiv (local) INTEGER array.
The dimension of *ipiv* must be ≥ *desca*(NB).
Contains pivot indices for local factorizations. Note that you *should not alter* the contents of this array between factorization and solve.

af (local)

REAL for psgbtrf
DOUBLE PRECISION for pdgbtrf
COMPLEX for pcgbtrf
DOUBLE COMPLEX for pzgbtrf.

Array, dimension (*laf*).
Auxiliary Fillin space. Fillin is created during the factorization routine p?gbtrf and this is stored in *af*.
Note that if a linear system is to be solved using p?gbtrs after the factorization routine, *af* must not be altered after the factorization.

work(1) — On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.

If *info*=0, the execution is successful.
info < 0:
if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.
info > 0:

If *info* = k ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not nonsingular, and the factorization was not completed. If *info* = k>NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?dbtrf

Computes the LU factorization of a n-by-n diagonally dominant-like banded distributed matrix.

Syntax

```
call psdbtrf ( n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info )
call pddbtrf ( n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info )
call pcdbtrf ( n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info )
call pzdbtrf ( n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info )
```

Description

The routine computes the *LU* factorization of a *n*-by-*n* real/complex diagonally dominant-like banded distributed matrix *A*(1:*n*, *ja*:*ja*+*n*-1) without pivoting.

Note that the resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

Input Parameters

n (global) INTEGER. The number of rows and columns in the distributed submatrix *A*(1:*n*, *ja*:*ja*+*n*-1); $n \ge 0$.

bwl (global) INTEGER. The number of sub-diagonals within the band of *A* ($0 \le bwl \le n-1$).

bwu (global) INTEGER. The number of super-diagonals within the band of *A* ($0 \le bwu \le n-1$).

a (local)

REAL for psdbtrf
DOUBLE PRECISION for pddbtrf
COMPLEX for pcdbtrf
DOUBLE COMPLEX for pzdbtrf.
Pointer into the local memory to an array of local dimension (*lld_a*, LOC_c(*ja*+*n*-1)).
Contains the local pieces of the *n*-by-*n* distributed banded matrix *A*(1:*n*, *ja*:*ja*+*n*-1) to be factored.

ja (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7;
else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

laf (local) INTEGER. The dimension of the array *af*.
Must be $laf \ge NB*(bwl+bwu)+6*(\max(bwl,bwu))^2$.

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work (local) Same type as *a*. Workspace array of dimension *lwork* .

lwork (local or global) `INTEGER`. The size of the *work* array, must be *lwork* ≥ $(\max(bwl,bwu))^2$. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

Output Parameters

a On exit, this array contains details of the factorization. Note that additional permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.

af (local)

`REAL` for `psdbtrf`
`DOUBLE PRECISION` for `pddbtrf`
`COMPLEX` for `pcdbtrf`
`DOUBLE COMPLEX` for `pzdbtrf`.

Array, dimension (*laf*).
Auxiliary Fillin space. Fillin is created during the factorization routine `p?dbtrf` and this is stored in *af*.
Note that if a linear system is to be solved using `p?dbtrs` after the factorization routine, *af* must not be altered after the factorization.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info (global) `INTEGER`.

If *info*=0, the execution is successful.
info < 0:
if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.
info > 0:

If *info* = k ≤ `NPROCS`, the submatrix stored on processor *info* and factored locally was not diagonally dominant-like, and the factorization was not completed. If *info* = k>`NPROCS`, the submatrix stored on processor *info*-`NPROCS` representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?potrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite distributed matrix.

Syntax

```
call pspotrf ( uplo, n, a, ia, ja, desca, info )
call pdpotrf ( uplo, n, a, ia, ja, desca, info )
call pcpotrf ( uplo, n, a, ia, ja, desca, info )
call pzpotrf ( uplo, n, a, ia, ja, desca, info )
```

Description

This routine computes the Cholesky factorization of a real symmetric or complex Hermitian positive-definite distributed *n*-by-*n* matrix A(*ia:ia+n-1*, *ja:ja+n-1*), denoted below as sub(A).

The factorization has the form

sub(A) = $U^H U$ if `uplo='U'`, or

sub(A) = LL^H if `uplo='L'`

where L is a lower triangular matrix and U is upper triangular.

Input Parameters

uplo (global) `CHARACTER*1`. Must be `'U'` or `'L'`.

Indicates whether the upper or lower triangular part of sub(A) is stored:

If *uplo* = `'U'`, the array *a* stores the upper triangular part of the matrix sub(A), and sub(A) is factored as $U^H U$.
If *uplo* = `'L'`, the array *a* stores the lower triangular part of the matrix sub(A), and sub(A) is factored as LL^H.

n (global) `INTEGER`. The order of the distributed submatrix sub(A) ($n \geq 0$).

a (local)

REAL for pspotrf
DOUBLE PRECISION for pdpotrf
COMPLEX for pcpotrf
DOUBLE COMPLEX for pzpotrf.

Pointer into the local memory to an array of dimension (*lld_a*, LOC_c(*ja*+*n*-*1*)).

On entry, this array contains the local pieces of the *n*-by-*n* symmetric/Hermitian distributed matrix sub(*A*) to be factored.

Depending on *uplo*, the array *a* contains either the upper or the lower triangular part of the matrix sub(*A*) (see *uplo*).

ia, ja (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

a The upper or lower triangular part of *a* is overwritten by the Cholesky factor *U* or *L*, as specified by *uplo*.

info (global) INTEGER.

If *info*=0, the execution is successful;
info < 0: if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.
If *info* = *k* >0, the leading minor of order *k*, *A*(*ia*:*ia*+*k*-*1*, *ja*:*ja*+*k*-*1*), is not positive-definite, and the factorization could not be completed.

p?pbtrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite banded distributed matrix.

Syntax

```
call pspbtrf ( uplo, n, bw, a, ja, desca, af, laf, work, lwork, info )
```

```
call pdpbtrf ( uplo, n, bw, a, ja, desca, af, laf, work, lwork, info )
call pcpbtrf ( uplo, n, bw, a, ja, desca, af, laf, work, lwork, info )
call pzpbtrf ( uplo, n, bw, a, ja, desca, af, laf, work, lwork, info )
```

Description

This routine computes the Cholesky factorization of an *n*-by-*n* real symmetric or complex Hermitian positive-definite banded distributed matrix A(1:*n*, *ja*:*ja*+*n*-1).

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

A(1:*n*, *ja*:*ja*+*n*-1) = $P\,U^H U\,P^T$,   if *uplo*='U', or

A(1:*n*, *ja*:*ja*+*n*-1) = $P\,L\,L^H\,P^T$,   if *uplo*='L',

where P is a permutation matrix and U and L are banded upper and lower triangular matrices, respectively.

Input Parameters

uplo (global) `CHARACTER*1`. Must be 'U' or 'L'.

If *uplo* = 'U', upper triangle of A(1:*n*, *ja*:*ja*+*n*-1) is stored;
If *uplo* = 'L', lower triangle of A(1:*n*, *ja*:*ja*+*n*-1) is stored.

n (global) `INTEGER`. The order of the distributed submatrix A(1:*n*, *ja*:*ja*+*n*-1) ($n \geq 0$).

bw (global) `INTEGER`. The number of superdiagonals of the distributed matrix if *uplo* = 'U', or the number of subdiagonals if *uplo* = 'U' ($bw \geq 0$).

a (local)

`REAL` for `pspbtrf`
`DOUBLE PRECISION` for `pdpbtrf`
`COMPLEX` for `pcpbtrf`
`DOUBLE COMPLEX` for `pzpbtrf`.

Pointer into the local memory to an array of dimension (*lld_a*, LOC_c(*ja*+*n*-1)).

On entry, this array contains the local pieces of the upper or lower triangle of the symmetric/Hermitian band distributed matrix A(1:*n*, *ja*:*ja*+*n*-1) to be factored.

ja (global) `INTEGER`. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7;
else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

laf (local) `INTEGER`. The dimension of the array *af*.
Must be *laf* ≥ (NB+2**bw*)**bw* .

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work (local) Same type as *a*. Workspace array of dimension *lwork* .

lwork (local or global) `INTEGER`. The size of the *work* array, must be *lwork* ≥ bw^2.

Output Parameters

a On exit, if *info*=0, contains the permuted triangular factor *U* or *L* from the Cholesky factorization of the band matrix *A*(*1:n*, *ja:ja+n-1*), as specified by *uplo*.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info (global) `INTEGER`.

If *info*=0, the execution is successful.
info < 0:
if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.
info > 0:

If *info* = k ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.
If *info* = k>NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?pttrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite tridiagonal distributed matrix.

Syntax

```
call pspttrf ( n, d, e, ja, desca, af, laf, work, lwork, info )
call pdpttrf ( n, d, e, ja, desca, af, laf, work, lwork, info )
call pcpttrf ( n, d, e, ja, desca, af, laf, work, lwork, info )
call pzpttrf ( n, d, e, ja, desca, af, laf, work, lwork, info )
```

Description

This routine computes the Cholesky factorization of an *n*-by-*n* real symmetric or complex Hermitian positive-definite tridiagonal distributed matrix A(1:*n*, *ja*:*ja*+*n*-1).

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

$A(1{:}n, ja{:}ja{+}n{-}1) = P\,L\,D\,L^H\,P^T$, or

$A(1{:}n, ja{:}ja{+}n{-}1) = P\,U^H\,D\,U\,P^T$,

where P is a permutation matrix, and U and L are tridiagonal upper and lower triangular matrices, respectively.

Input Parameters

n (global) `INTEGER`. The order of the distributed submatrix A(1:*n*, *ja*:*ja*+*n*-1) ($n \geq 0$).

d, e (local)

`REAL` for `pspttrf`
`DOUBLE PRECISION` for `pdpttrf`
`COMPLEX` for `pcpttrf`
`DOUBLE COMPLEX` for `pzpttrf`.

Pointers into the local memory to arrays of dimension (*desca*(*nb_*)) each.

On entry, the array *d* contains the local part of the global vector storing the main diagonal of the distributed matrix *A*.

On entry, the array *e* contains the local part of the global vector storing the upper diagonal of the distributed matrix *A*.

ja (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7;
else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

laf (local) INTEGER. The dimension of the array *af*.
Must be *laf* ≥ NB+2 .

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work (local) Same type as *d* and *e*. Workspace array of dimension *lwork* .

lwork (local or global) INTEGER. The size of the *work* array, must be at least *lwork* ≥ 8*NPCOL.

Output Parameters

d, e On exit, overwritten by the details of the factorization.

af (local)

REAL for pspttrf
DOUBLE PRECISION for pdpttrf
COMPLEX for pcpttrf
DOUBLE COMPLEX for pzpttrf.

Array, dimension (*laf*).
Auxiliary Fillin space. Fillin is created during the factorization routine p?pttrf and this is stored in *af*.
Note that if a linear system is to be solved using p?pttrs after the factorization routine, *af* must not be altered.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.

If `info`=0, the execution is successful.
`info` < 0:
if the `i`th argument is an array and the `j`th entry had an illegal value, then `info` = `-(i*100+j)`; if the `i`th argument is a scalar and had an illegal value, then `info` = `-i`.
`info` > 0:

If `info` = k ≤ NPROCS, the submatrix stored on processor `info` and factored locally was not positive definite, and the factorization was not completed.
If `info` = k>NPROCS, the submatrix stored on processor `info`-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?dttrf

Computes the LU factorization of a diagonally dominant-like tridiagonal distributed matrix.

Syntax

```
call psdttrf ( n, dl, d, du, ja, desca, af, laf, work, lwork, info )
call pddttrf ( n, dl, d, du, ja, desca, af, laf, work, lwork, info )
call pcdttrf ( n, dl, d, du, ja, desca, af, laf, work, lwork, info )
call pzdttrf ( n, dl, d, du, ja, desca, af, laf, work, lwork, info )
```

Description

This routine computes the *LU* factorization of an `n`-by-`n` real/complex diagonally dominant-like tridiagonal distributed matrix *A*(`1:n`, `ja:ja+n-1`) without pivoting for stability.

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

$$A(1:n, ja:ja+n-1) = P\,L\,U\,P^T,$$

where *P* is a permutation matrix, and *L* and *U* are banded lower and upper triangular matrices, respectively.

Input Parameters

n (global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $A(1:n, ja:ja+n-1)$ $(n \geq 0)$.

dl,d,du (local)

REAL for pspttrf
DOUBLE PRECISION for pdpttrf
COMPLEX for pcpttrf
DOUBLE COMPLEX for pzpttrf.

Pointers to the local arrays of dimension (*desca*(*nb_*)) each.

On entry, the array *dl* contains the local part of the global vector storing the subdiagonal elements of the matrix. Globally, *dl*(1) is not referenced, and *dl* must be aligned with *d*.

On entry, the array *d* contains the local part of the global vector storing the diagonal elements of the matrix.

On entry, the array *du* contains the local part of the global vector storing the super-diagonal elements of the matrix. *du*(*n*) is not referenced, and *du* must be aligned with *d*.

ja (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7;
else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

laf (local) INTEGER. The dimension of the array *af*.
Must be *laf* ≥ 2*(NB+2) .

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work (local) Same type as *d*. Workspace array of dimension *lwork* .

lwork (local or global) INTEGER. The size of the *work* array, must be at least *lwork* ≥ 8*NPCOL.

Output Parameters

dl,d,du On exit, overwritten by the information containing the factors of the matrix.

af (local)

REAL for psdttrf
DOUBLE PRECISION for pddttrf
COMPLEX for pcdttrf
DOUBLE COMPLEX for pzdttrf.

Array, dimension (*laf*).
Auxiliary Fillin space. Fillin is created during the factorization routine p?dttrf and this is stored in *af*.
Note that if a linear system is to be solved using p?dttrs after the factorization routine, *af* must not be altered.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.

If *info*=0, the execution is successful.
info < 0:
if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.
info > 0:

If *info* = k ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not diagonally dominant-like, and the factorization was not completed.
If *info* = k>NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

Routines for Solving Systems of Linear Equations

This section describes the ScaLAPACK routines for solving systems of linear equations. Before calling most of these routines, you need to factorize the matrix of your system of equations (see Routines for Matrix Factorization in this chapter). However, the factorization is not necessary if your system of equations has a triangular matrix.

p?getrs

Solves a system of distributed linear equations with a general square matrix, using the LU factorization computed by `p?getrf`.

Syntax

```
call psgetrs (trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb,
    info)
call pdgetrs (trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb,
    info)
call pcgetrs (trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb,
    info)
call pzgetrs (trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb,
    info)
```

Description

This routine solves a system of distributed linear equations with a general *n*-by-*n* distributed matrix sub(*A*) = *A*(*ia:ia+n-1*, *ja:ja+n-1*) using the *LU* factorization computed by `p?getrf`.

The system has one of the following forms specified by *trans*:

sub(*A*)**X* = sub(*B*) (no transpose),

$\text{sub}(A)^T * X = \text{sub}(B)$ (transpose),

$\text{sub}(A)^H * X = \text{sub}(B)$ (conjugate transpose),

where sub(*B*) = *B*(*ib:ib+n-1*, *jb:jb+nrhs-1*) .

Before calling this routine, you must call `p?getrf` to compute the *LU* factorization of sub(*A*).

Input Parameters

trans (global) `CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`.

Indicates the form of the equations:

If *trans* = `'N'`, then sub(*A*)**X* = sub(*B*) is solved for *X*.

If *trans* = `'T'`, then $\text{sub}(A)^T * X = \text{sub}(B)$ is solved for *X*.

If *trans* = `'C'`, then $\text{sub}(A)^H * X = \text{sub}(B)$ is solved for *X*.

n (global) INTEGER. The number of linear equations; the order of the submatrix sub(*A*) ($n \geq 0$).

nrhs (global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix sub(*B*) ($nrhs \geq 0$).

a, b (global)

REAL for psgetrs
DOUBLE PRECISION for pdgetrs
COMPLEX for pcgetrs
DOUBLE COMPLEX for pzgetrs.
Pointers into the local memory to arrays of local dimension *a*(*lld_a*, LOC_c(*ja*+*n*-1)) and *b*(*lld_b*, LOC_c(*jb*+*nrhs*-1)), respectively.
On entry, the array *a* contains the local pieces of the factors *L* and *U* from the factorization sub(*A*) = *PLU* ; the unit diagonal elements of *L* are not stored.
On entry, the array *b* contains the right hand sides sub(*B*).

ia,ja (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

ipiv (local) INTEGER array.
The dimension of *ipiv* is (LOC_r(*m_a*)+ *mb_a*).
This array contains contains the pivoting information: local row i of the matrix was interchanged with the global row *ipiv*(i).
This array is tied to the distributed matrix *A*.

ib,jb (global) INTEGER. The row and column indices in the global array *B* indicating the first row and the first column of the submatrix sub(*B*), respectively.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

Output Parameters

b On exit, overwritten by the solution distributed matrix *X*.

info INTEGER. If *info*=0, the execution is successful.
info < 0:

if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.

p?gbtrs

Solves a system of distributed linear equations with a general band matrix, using the LU factorization computed by `p?gbtrf`.

Syntax

```
call psgbtrs (trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb,
    af, laf, work, lwork, info)
call pdgbtrs (trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb,
    af, laf, work, lwork, info)
call pcgbtrs (trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb,
    af, laf, work, lwork, info)
call pzgbtrs (trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb,
    af, laf, work, lwork, info)
```

Description

This routine solves a system of distributed linear equations with a general band distributed matrix sub(*A*) = *A*(1:*n*, *ja*:*ja*+*n*-1) using the *LU* factorization computed by `p?gbtrf`.

The system has one of the following forms specified by *trans*:

sub(*A*)**X* = sub(*B*) (no transpose),

sub$(A)^T$**X* = sub(*B*) (transpose),

sub$(A)^H$ **X* = sub(*B*) (conjugate transpose),

where sub(*B*) = *B*(*ib*:*ib*+*n*-1, 1:*nrhs*) .

Before calling this routine, you must call `p?gbtrf` to compute the *LU* factorization of sub(*A*).

Input Parameters

trans (global) `CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`.

Indicates the form of the equations:

	If *trans* = 'N', then sub(A)*X = sub(B) is solved for X.
	If *trans* = 'T', then sub(A)^T*X = sub(B) is solved for X.
	If *trans* = 'C', then sub(A)^H *X = sub(B) is solved for X.
n	(global) INTEGER. The number of linear equations; the order of the distributed submatrix sub(A) ($n \geq 0$).
bwl	(global) INTEGER. The number of sub-diagonals within the band of A ($0 \leq bwl \leq n-1$).
bwu	(global) INTEGER. The number of super-diagonals within the band of A ($0 \leq bwu \leq n-1$).
nrhs	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix sub(B) ($nrhs \geq 0$).
a, b	(global) REAL for psgbtrs DOUBLE PRECISION for pdgbtrs COMPLEX for pcgbtrs DOUBLE COMPLEX for pzgbtrs. Pointers into the local memory to arrays of local dimension *a*(*lld_a*, LOC_c(*ja*+*n*-1)) and *b*(*lld_b*, LOC_c(*nrhs*)), respectively. The array *a* contains details of the LU factorization of the distributed band matrix A. On entry, the array *b* contains the local pieces of the right hand sides B(*ib*:*ib*+*n*-1, 1:*nrhs*).
ja	(global) INTEGER. The index in the global array A that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix A. If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7; else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.
ib	(global) INTEGER. The index in the global array A that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
descb	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix A. If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7; else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

laf (local) INTEGER. The dimension of the array *af*.
Must be *laf* ≥ NB*(*bwl*+*bwu*)+6*(*bwl*+*bwu*)*(*bwl*+2**bwu*) .

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work (local) Same type as *a*. Workspace array of dimension *lwork* .

lwork (local or global) INTEGER. The size of the *work* array, must be at least *lwork* ≥ *nrhs**(NB+2**bwl*+4**bwu*) .

Output Parameters

ipiv (local) INTEGER array.
The dimension of *ipiv* must be ≥ *desca*(NB).
Contains pivot indices for local factorizations. Note that you should not alter the contents of this array between factorization and solve.

b On exit, overwritten by the local pieces of the solution distributed matrix *X*.

af (local)

REAL for psgbtrs
DOUBLE PRECISION for pdgbtrs
COMPLEX for pcgbtrs
DOUBLE COMPLEX for pzgbtrs.

Array, dimension (*laf*).
Auxiliary Fillin space. Fillin is created during the factorization routine p?gbtrf and this is stored in *af*.
Note that if a linear system is to be solved using p?gbtrs after the factorization routine, *af* must not be altered after the factorization.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info INTEGER. If *info*=0, the execution is successful.
info < 0:

if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.

p?potrs

Solves a system of linear equations with a Cholesky-factored symmetric/Hermitian distributed positive-definite matrix.

Syntax

```
call pspotrs ( uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info )
call pdpotrs ( uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info )
call pcpotrs ( uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info )
call pzpotrs ( uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info )
```

Description

The routine `p?potrs` solves for X a system of distributed linear equations in the form:

$$\mathrm{sub}(A)*X = \mathrm{sub}(B) ,$$

where $\mathrm{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ is an n-by-n real symmetric or complex Hermitian positive definite distributed matrix, and $\mathrm{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, jb:jb+nrhs-1)$.
This routine uses Cholesky factorization

$$\mathrm{sub}(A) = U^H U \ \text{ or } \ \mathrm{sub}(A) = L L^H$$

computed by `p?potrf`.

Input Parameters

uplo (global) `CHARACTER*1`. Must be `'U'` or `'L'`.

If *uplo* = `'U'`, upper triangle of sub(A) is stored;
If *uplo* = `'L'`, lower triangle of sub(A) is stored.

n (global) `INTEGER`. The order of the distributed submatrix sub(A) ($n \geq 0$).

nrhs (global) `INTEGER`. The number of right hand sides; the number of columns of the distributed submatrix sub(B) ($nrhs \geq 0$).

a, b (local)

	`REAL` for `pspotrs` `DOUBLE PRECISION` for `pdpotrs` `COMPLEX` for `pcpotrs` `DOUBLE COMPLEX` for `pzpotrs`. Pointers into the local memory to arrays of local dimension `a(lld_a,` LOC_c`(ja+n-1))` and `b(lld_b,` LOC_c`(jb+nrhs-1))`, respectively. The array *a* contains the factors *L* or *U* from the Cholesky factorization sub(*A*) = $L L^H$ or sub(*A*) = $U^H U$, as computed by `p?potrf`. On entry, the array *b* contains the local pieces of the right hand sides sub(*B*).
ia,ja	(global) `INTEGER`. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.
desca	(global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
ib,jb	(global) `INTEGER`. The row and column indices in the global array *B* indicating the first row and the first column of the submatrix sub(*B*), respectively.
descb	(global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

Output Parameters

b	Overwritten by the local pieces of the solution matrix *X*.
info	`INTEGER`. If *info*=0, the execution is successful. *info* < 0: if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = `-(i*100+j)`; if the *i*th argument is a scalar and had an illegal value, then *info* = *-i*.

p?pbtrs

Solves a system of linear equations with a Cholesky-factored symmetric/Hermitian positive-definite band matrix.

Syntax

```
call pspbtrs ( uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf,
    work, lwork, info )
call pdpbtrs ( uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf,
    work, lwork, info )
call pcpbtrs ( uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf,
    work, lwork, info )
call pzpbtrs ( uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf,
    work, lwork, info )
```

Description

The routine `p?pbtrs` solves for X a system of distributed linear equations in the form:

$$\mathrm{sub}(A)*X = \mathrm{sub}(B) ,$$

where sub(A) = A(*1:n*, *ja:ja+n-1*) is an *n*-by-*n* real symmetric or complex Hermitian positive definite distributed band matrix, and sub(B) denotes the distributed matrix B(*ib:ib+n-1*, *1:nrhs*).
This routine uses Cholesky factorization

$$\mathrm{sub}(A) = P\,U^H\,U\,P^T \quad \text{or} \quad \mathrm{sub}(A) = P\,L\,L^H\,P^T$$

computed by `p?pbtrf`.

Input Parameters

uplo (global) `CHARACTER*1`. Must be `'U'` or `'L'`.

If *uplo* = `'U'`, upper triangle of sub(A) is stored;
If *uplo* = `'L'`, lower triangle of sub(A) is stored.

n (global) `INTEGER`. The order of the distributed submatrix sub(A) ($n \geq 0$).

bw (global) `INTEGER`. The number of superdiagonals of the distributed matrix if *uplo* = `'U'`, or the number of subdiagonals if *uplo* = `'L'` ($bw \geq 0$).

nrhs (global) `INTEGER`. The number of right hand sides; the number of columns of the distributed submatrix sub(*B*) (*nrhs* ≥ 0).

a, b (local)

`REAL` for `pspbtrs`
`DOUBLE PRECISION` for `pdpbtrs`
`COMPLEX` for `pcpbtrs`
`DOUBLE COMPLEX` for `pzpbtrs`.
Pointers into the local memory to arrays of local dimension *a*(*lld_a*, LOC_c(*ja*+*n*-1)) and *b*(*lld_b*, LOC_c(*nrhs*-1)), respectively.
The array *a* contains the permuted triangular factor *U* or *L* from the Cholesky factorization sub(*A*) = $P U^H U P^T$ or sub(*A*) = $P L L^H P^T$ of the band matrix *A*, as returned by `p?pbtrf`.

On entry, the array *b* contains the local pieces of the *n*-by-*nrhs* right hand side distributed matrix sub(*B*).

ja (global) `INTEGER`. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7;
else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

ib (global) `INTEGER`. The row index in the global array *B* indicating the first row of the submatrix sub(*B*).

descb (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

If *descb*(*dtype_*) = 502, then *dlen_* ≥ 7;
else if *descb*(*dtype_*) = 1, then *dlen_* ≥ 9.

af, work (local) Arrays, same type as *a*.
The array *af* is of dimension (*laf*). It contains auxiliary Fillin space. Fillin is created during the factorization routine `p?dbtrf` and this is stored in *af*.

The array *work* is a workspace array of dimension *lwork*.

laf (local) `INTEGER`. The dimension of the array *af*.
Must be *laf* ≥ *nrhs***bw* .

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

lwork	(local or global) INTEGER. The size of the array *work*, must be at least $lwork \geq bw^2$.

Output Parameters

b	On exit, if *info*=0, this array contains the local pieces of the *n*-by-*nrhs* solution distributed matrix *X*.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	INTEGER. If *info*=0, the execution is successful. *info* < 0: if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.

p?pttrs

Solves a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal distributed matrix using the factorization computed by p?pttrf .

Syntax

```
call pspttrs ( n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work,
    lwork, info )
call pdpttrs ( n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work,
    lwork, info )
call pcpttrs ( uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf,
    work, lwork, info )
call pzpttrs ( uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf,
    work, lwork, info )
```

Description

The routine p?pttrs solves for *X* a system of distributed linear equations in the form:

$$\mathrm{sub}(A)*X = \mathrm{sub}(B) ,$$

where sub(A) = A(`1:n`, `ja:ja+n-1`) is an *n*-by-*n* real symmetric or complex Hermitian positive definite tridiagonal distributed matrix, and sub(B) denotes the distributed matrix B(`ib:ib+n-1`, `1:nrhs`).
This routine uses the factorization

$$\text{sub}(A) = P\,L\,D\,L^H\,P^T \quad \text{or} \quad \text{sub}(A) = P\,U^H D\,U\,P^T$$

computed by `p?pttrf`.

Input Parameters

uplo (global, used in complex flavors only)
`CHARACTER*1`. Must be `'U'` or `'L'`.

If *uplo* = `'U'`, upper triangle of sub(A) is stored;
If *uplo* = `'L'`, lower triangle of sub(A) is stored.

n (global) `INTEGER`. The order of the distributed submatrix sub(A) ($n \geq 0$).

nrhs (global) `INTEGER`. The number of right hand sides; the number of columns of the distributed submatrix sub(B) ($nrhs \geq 0$).

d, e (local)

`REAL` for `psptrs`
`DOUBLE PRECISION` for `pdpttrs`
`COMPLEX` for `pcpttrs`
`DOUBLE COMPLEX` for `pzpttrs`.

Pointers into the local memory to arrays of dimension (*desca*(*nb_*)) each.

These arrays contain details of the factorization as returned by `p?pttrf`

ja (global) `INTEGER`. The index in the global array A that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).

desca (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix A.
If *desca*(*dtype_*) = 501 or 502, then *dlen_* ≥ 7;
else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

b (local) Same type as *d, e*.

Pointer into the local memory to an array of local dimension *b*(*lld_b*, LOC_c(*nrhs*)).
On entry, the array *b* contains the local pieces of the *n*-by-*nrhs* right hand side distributed matrix sub(B).

ib (global) INTEGER. The row index in the global array *B* that points to the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

If *descb*(*dtype_*) = 502, then *dlen_* ≥ 7;
else if *descb*(*dtype_*) = 1, then *dlen_* ≥ 9.

af, work (local)
REAL for psptters
DOUBLE PRECISION for pdpttrs
COMPLEX for pcpttrs
DOUBLE COMPLEX for pzpttrs.

Arrays of dimension (*laf*) and (*lwork*), respectively
The array *af* contains auxiliary Fillin space. Fillin is created during the factorization routine p?pttrf and this is stored in *af*.

The array *work* is a workspace array.

laf (local) INTEGER. The dimension of the array *af*.
Must be *laf* ≥ NB+2 .

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

lwork (local or global) INTEGER. The size of the array *work*, must be at least *lwork* ≥ (10+2*min(100,*nrhs*))*NPCOL+4***nrhs*.

Output Parameters

b On exit, this array contains the local pieces of the solution distributed matrix *X*.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info INTEGER. If *info*=0, the execution is successful.
info < 0:

if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.

p?dttrs

Solves a system of linear equations with a diagonally dominant-like tridiagonal distributed matrix using the factorization computed by `p?dttrf` *.*

Syntax

```
call psdttrs ( trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af,
    laf, work, lwork, info )
call pddttrs ( trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af,
    laf, work, lwork, info )
call pcdttrs ( trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af,
    laf, work, lwork, info )
call pzdttrs ( trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af,
    laf, work, lwork, info )
```

Description

The routine `p?dttrs` solves for X one of the systems of equations:

$$\text{sub}(A)*X = \text{sub}(B)\ ,$$

$$(\text{sub}(A))^T*X = \text{sub}(B)\ ,\ \text{or}$$

$$(\text{sub}(A))^H*X = \text{sub}(B)\ ,$$

where sub(A) = A(*1:n*, *ja:ja+n-1*) is a diagonally dominant-like tridiagonal distributed matrix, and sub(B) denotes the distributed matrix B(*ib:ib+n-1*, *1:nrhs*).
This routine uses the LU factorization computed by `p?dttrf`.

Input Parameters

trans (global) `CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`.

Indicates the form of the equations:

If *trans* = `'N'`, then sub(A)*X = sub(B) is solved for X.

If *trans* = `'T'`, then sub(A)^T*X = sub(B) is solved for X.

If *trans* = `'C'`, then sub(A)^H *X = sub(B) is solved for X.

n (global) `INTEGER`. The order of the distributed submatrix sub(A) ($n \geq 0$).

nrhs (global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix sub(*B*) ($nrhs \geq 0$).

dl,d,du (local)

REAL for psdttrs
DOUBLE PRECISION for pddttrs
COMPLEX for pcdttrs
DOUBLE COMPLEX for pzdttrs.

Pointers to the local arrays of dimension (*desca*(*nb_*)) each.

On entry, these arrays contain details of the factorization. Globally, *dl*(1) and *du*(*n*) are not referenced; *dl* and *du* must be aligned with *d*.

ja (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
If *desca*(*dtype_*) = 501 or 502, then *dlen_* ≥ 7;
else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

b (local) Same type as *d*.

Pointer into the local memory to an array of local dimension *b*(*lld_b*, LOC_c(*nrhs*)).
On entry, the array *b* contains the local pieces of the *n*-by-*nrhs* right hand side distributed matrix sub(*B*).

ib (global) INTEGER. The row index in the global array *B* that points to the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

If *descb*(*dtype_*) = 502, then *dlen_* ≥ 7;
else if *descb*(*dtype_*) = 1, then *dlen_* ≥ 9.

af, work (local)
REAL for psdttrs
DOUBLE PRECISION for pddttrs
COMPLEX for pcdttrs
DOUBLE COMPLEX for pzdttrs.

Arrays of dimension (*laf*) and (*lwork*), respectively.
The array *af* contains auxiliary Fillin space. Fillin is created during the factorization routine `p?dttrf` and this is stored in *af*. If a linear system is to be solved using `p?dttrs` after the factorization routine, *af* must not be altered.

The array *work* is a workspace array.

laf (local) `INTEGER`. The dimension of the array *af*.
Must be *laf* ≥ NB*(*bwl*+*bwu*)+6*(*bwl*+*bwu*)*(*bwl*+2*bwu*) .

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

lwork (local or global) `INTEGER`. The size of the array *work*, must be at least *lwork* ≥ `10*NPCOL+4*`*nrhs*.

Output Parameters

b On exit, this array contains the local pieces of the solution distributed matrix *X*.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info `INTEGER`. If *info*=0, the execution is successful.
info < 0:

if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.

p?dbtrs

Solves a system of linear equations with a diagonally dominant-like banded distributed matrix using the factorization computed by `p?dbtrf`.

Syntax

```
call psdbtrs ( trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af,
    laf, work, lwork, info )
```

```
call pddbtrs ( trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af,
    laf, work, lwork, info )
call pcdbtrs ( trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af,
    laf, work, lwork, info )
call pzdbtrs ( trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af,
    laf, work, lwork, info )
```

Description

The routine `p?dbtrs` solves for X one of the systems of equations:

$$\text{sub}(A)*X = \text{sub}(B)\ ,$$

$$(\text{sub}(A))^T*X = \text{sub}(B)\ ,\ \text{or}$$

$$(\text{sub}(A))^H*X = \text{sub}(B)\ ,$$

where sub(A) = A(*1:n*, *ja:ja+n-1*) is a diagonally dominant-like banded distributed matrix, and sub(B) denotes the distributed matrix B(*ib:ib+n-1*, *1:nrhs*).
This routine uses the LU factorization computed by `p?dbtrf`.

Input Parameters

trans (global) `CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`.

Indicates the form of the equations:

If *trans* = `'N'`, then sub(A)*X = sub(B) is solved for X.

If *trans* = `'T'`, then sub(A)^T*X = sub(B) is solved for X.

If *trans* = `'C'`, then sub(A)^H *X = sub(B) is solved for X.

n (global) `INTEGER`. The order of the distributed submatrix sub(A) ($n \geq 0$).

bwl (global) `INTEGER`. The number of subdiagonals within the band of A ($0 \leq bwl \leq n-1$).

bwu (global) `INTEGER`. The number of superdiagonals within the band of A ($0 \leq bwu \leq n-1$).

nrhs (global) `INTEGER`. The number of right hand sides; the number of columns of the distributed submatrix sub(B) ($nrhs \geq 0$).

a, b (local)

REAL for psdbtrs
DOUBLE PRECISION for pddbtrs
COMPLEX for pcdbtrs
DOUBLE COMPLEX for pzdbtrs.

Pointers into the local memory to arrays of local dimension a(*lld_a*, LOC_c(*ja*+*n*-1)) and b(*lld_b*, LOC_c(*nrhs*)), respectively.

On entry, the array *a* contains details of the *LU* factorization of the band matrix *A*, as computed by p?dbtrf.

On entry, the array *b* contains the local pieces of the right hand side distributed matrix sub(*B*).

ja (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7;
else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

ib (global) INTEGER. The row index in the global array *B* that points to the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

If *descb*(*dtype_*) = 502, then *dlen_* ≥ 7;
else if *descb*(*dtype_*) = 1, then *dlen_* ≥ 9.

af, work (local)
REAL for psdbtrs
DOUBLE PRECISION for pddbtrs
COMPLEX for pcdbtrs
DOUBLE COMPLEX for pzdbtrs.

Arrays of dimension (*laf*) and (*lwork*), respectively
The array *af* contains auxiliary Fillin space. Fillin is created during the factorization routine p?dbtrf and this is stored in *af*.

The array *work* is a workspace array.

laf (local) INTEGER. The dimension of the array *af*.
Must be *laf* ≥ NB*(*bwl*+*bwu*)+6*(max(*bwl*,*bwu*))^2 .

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

lwork — (local or global) INTEGER. The size of the array *work*, must be at least $lwork \geq (\max(bwl,bwu))^2$.

Output Parameters

b — On exit, this array contains the local pieces of the solution distributed matrix *X*.

work(1) — On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — INTEGER. If *info*=0, the execution is successful.
info < 0:

if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.

p?trtrs

Solves a system of linear equations with a triangular distributed matrix.

Syntax

```
call pstrtrs (uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb,
    descb, info)
call pdtrtrs (uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb,
    descb, info)
call pctrtrs (uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb,
    descb, info)
call pztrtrs (uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb,
    descb, info)
```

Description

This routine solves for *X* one of the following systems of linear equations:

$$\text{sub}(A)*X = \text{sub}(B),$$

$(\text{sub}(A))^T * X = \text{sub}(B)$, or

$(\text{sub}(A))^H * X = \text{sub}(B)$,

where sub(A) = A(`ia:ia+n-1`, `ja:ja+n-1`) is a triangular distributed matrix of order *n*, and sub(B) denotes the distributed matrix B(`ib:ib+n-1`, `jb:jb+nrhs-1`).
A check is made to verify that sub(A) is nonsingular.

Input Parameters

uplo (global) `CHARACTER*1`. Must be `'U'` or `'L'`.

Indicates whether sub(A) is upper or lower triangular:

If *uplo* = `'U'`, then sub(A) is upper triangular.
If *uplo* = `'L'`, then sub(A) is lower triangular.

trans (global) `CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`.

Indicates the form of the equations:

If *trans* = `'N'`, then sub(A)*X = sub(B) is solved for X.

If *trans* = `'T'`, then $\text{sub}(A)^T * X$ = sub(B) is solved for X.

If *trans* = `'C'`, then $\text{sub}(A)^H * X$ = sub(B) is solved for X.

diag (global) `CHARACTER*1`. Must be `'N'` or `'U'`.

If *diag* = `'N'`, then sub(A) is not a unit triangular matrix.

If *diag* = `'U'`, then sub(A) is unit triangular.

n (global) `INTEGER`. The order of the distributed submatrix sub(A) ($n \geq 0$).

nrhs (global) `INTEGER`. The number of right-hand sides; i.e., the number of columns of the distributed matrix sub(B) ($nrhs \geq 0$).

a, b (local)
`REAL` for `pstrtrs`
`DOUBLE PRECISION` for `pdtrtrs`
`COMPLEX` for `pctrtrs`
`DOUBLE COMPLEX` for `pztrtrs`.

Pointers into the local memory to arrays of local dimension `a(`*lld_a*, LOC_c(*ja*+*n*-1)`)` and `b(`*lld_b*, LOC_c(*jb*+*nrhs*-1)`)`, respectively.

The array *a* contains the local pieces of the distributed triangular matrix sub(A).
If *uplo* = `'U'`, the leading *n*-by-*n* upper triangular part of sub(A) contains the

upper triangular matrix, and the strictly lower triangular part of sub(*A*) is not referenced.
If `uplo` = `'L'`, the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular matrix, and the strictly upper triangular part of sub(*A*) is not referenced.
If `diag` = `'U'`, the diagonal elements of sub(*A*) are also not referenced and are assumed to be 1.

On entry, the array *b* contains the local pieces of the right hand side distributed matrix sub(*B*).

ia, ja (global) `INTEGER`. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

ib, jb (global) `INTEGER`. The row and column indices in the global array *B* indicating the first row and the first column of the submatrix sub(*B*), respectively.

descb (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

Output Parameters

b On exit, if *info*=0, sub(*B*) is overwritten by the solution matrix *X*.

info `INTEGER`. If *info*=0, the execution is successful.

info < 0:

if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = `-(i*100+j)`; if the *i*th argument is a scalar and had an illegal value, then *info* = `-i`;

info > 0:

If *info* = *i*, the *i*th diagonal element of sub(*A*) is zero, indicating that the submatrix is singular and the solutions *X* have not been computed.

Routines for Estimating the Condition Number

This section describes the ScaLAPACK routines for estimating the condition number of a matrix. The condition number is used for analyzing the errors in the solution of a system of linear equations. Since the condition number may be arbitrarily large when the matrix is nearly singular, the routines actually compute the *reciprocal* condition number.

p?gecon

Estimates the reciprocal of the condition number of a general distributed matrix in either the 1-norm or the infinity-norm.

Syntax

```
call psgecon ( norm, n, a, ia, ja, desca, anorm, rcond, work, lwork,
    iwork, liwork, info )
call pdgecon ( norm, n, a, ia, ja, desca, anorm, rcond, work, lwork,
    iwork, liwork, info )
call pcgecon ( norm, n, a, ia, ja, desca, anorm, rcond, work, lwork,
    rwork, lrwork, info )
call pzgecon ( norm, n, a, ia, ja, desca, anorm, rcond, work, lwork,
    rwork, lrwork, info )
```

Description

This routine estimates the reciprocal of the condition number of a general distributed real/complex matrix sub(A) = A(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) in either the 1-norm or infinity-norm, using the LU factorization computed by `p?getrf`.

An estimate is obtained for $\|(\mathrm{sub}(A))^{-1}\|$, and the reciprocal of the condition number is computed as

$$rcond = \frac{1}{\|sub(A)\| \times \|(sub(A))^{-1}\|}$$

Input Parameters

norm (global) `CHARACTER*1`. Must be `'1'` or `'O'` or `'I'`.

Specifies whether the 1-norm condition number or the infinity-norm condition number is required.

If *norm* = '1' or 'O', then the 1-norm is used;

If *norm* = 'I', then the infinity-norm is used.

n (global) INTEGER. The order of the distributed submatrix sub(*A*) ($n \geq 0$).

a (local)
REAL for psgecon
DOUBLE PRECISION for pdgecon
COMPLEX for pcgecon
DOUBLE COMPLEX for pzgecon.

Pointer into the local memory to an array of dimension *a*(*lld_a*, LOC_c(*ja*+*n*-1)).

The array *a* contains the local pieces of the factors *L* and *U* from the factorization sub(*A*) = *P L U*; the unit diagonal elements of *L* are not stored.

ia,ja (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

anorm (global) REAL for single precision flavors,
DOUBLE PRECISION for double precision flavors.
If *norm* = '1' or 'O', the 1-norm of the original distributed matrix sub(*A*);

If *norm* = 'I', the infinity-norm of the original distributed matrix sub(*A*).

work (local)
REAL for psgecon
DOUBLE PRECISION for pdgecon
COMPLEX for pcgecon
DOUBLE COMPLEX for pzgecon.

The array *work* of dimension (*lwork*) is a workspace array.

lwork (local or global) INTEGER. The dimension of the array *work*.
For real flavors:
lwork must be at least
lwork ≥ 2*LOC_r(*n*+mod(*ia*-1,*mb_a*))+

$2*LOC_c(n+mod(ja-1,nb\_a))+$
max(2, max(*nb_a**max(1, ceil(NPROW-1,NPCOL)),
$LOC_c(n+mod(ja-1,nb_a))+nb_a*$max(1, ceil(NPCOL-1, NPROW))).

For complex flavors:
lwork must be at least
$lwork \geq 2*LOC_r(n+mod(ia-1,mb\_a))+$
max(2, max(*nb_a**ceil(NPROW-1, NPCOL),
$LOC_c(n+mod(ja-1,nb_a))+ nb_a*$ceil(NPCOL-1, NPROW))).

LOC_r and LOC_c values can be computed using the ScaLAPACK tool function numroc; NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

iwork (local) INTEGER.
Workspace array, DIMENSION (*liwork*). Used in real flavors only.

liwork (local or global) INTEGER.
The dimension of the array *iwork*; used in real flavors only. Must be at least
$liwork \geq LOC_r(n+mod(ia-1,mb\_a))$.

rwork (local) REAL for pcgecon
DOUBLE PRECISION for pzgecon
Workspace array, DIMENSION (*lrwork*). Used in complex flavors only.

lrwork (local or global) INTEGER.
The dimension of the array *rwork*; used in complex flavors only. Must be at least
$lrwork \geq 2*LOC_c(n+mod(ja-1,nb\_a))$.

Output Parameters

rcond (global) REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
The reciprocal of the condition number of the distributed matrix sub(*A*). See Description.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

iwork(1) On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).

rwork(1) On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).

info (global) `INTEGER`. If *info*=0, the execution is successful.

info < 0:

if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = `-(i*100+j)`; if the *i*th argument is a scalar and had an illegal value, then *info* = *-i*.

p?pocon

Estimates the reciprocal of the condition number (in the 1 - norm) of a symmetric / Hermitian positive-definite distributed matrix.

Syntax

```
call pspocon ( uplo, n, a, ia, ja, desca, anorm, rcond, work, lwork,
    iwork, liwork, info )
call pdpocon ( uplo, n, a, ia, ja, desca, anorm, rcond, work, lwork,
    iwork, liwork, info )
call pcpocon ( uplo, n, a, ia, ja, desca, anorm, rcond, work, lwork,
    rwork, lrwork, info )
call pzpocon ( uplo, n, a, ia, ja, desca, anorm, rcond, work, lwork,
    rwork, lrwork, info )
```

Description

This routine estimates the reciprocal of the condition number (in the 1 - norm) of a real symmetric or complex Hermitian positive definite distributed matrix sub(A) = A(*ia:ia+n-1*, *ja:ja+n-1*), using the Cholesky factorization sub(A) = $U^H U$ or sub(A) = LL^H computed by `p?potrf`.

An estimate is obtained for $\|(\mathrm{sub}(A))^{-1}\|$, and the reciprocal of the condition number is computed as

$$rcond = \frac{1}{\|sub(A)\| \times \|(sub(A))^{-1}\|}$$

Input Parameters

uplo (global) `CHARACTER*1`. Must be `'U'` or `'L'`.

Specifies whether the factor stored in sub(A) is upper or lower triangular.

If *uplo* = 'U', sub(A) stores the upper triangular factor U of the Cholesky factorization sub(A) = $U^H U$.
If *uplo* = 'L', sub(A) stores the lower triangular factor L of the Cholesky factorization sub(A) = LL^H.

n (global) INTEGER. The order of the distributed submatrix sub(A) ($n \geq 0$).

a (local)
REAL for pspocon
DOUBLE PRECISION for pdpocon
COMPLEX for pcpocon
DOUBLE COMPLEX for pzpocon.

Pointer into the local memory to an array of dimension *a*(*lld_a*, LOC_c(*ja*+*n*-1)).

The array *a* contains the local pieces of the factors L or U from the Cholesky factorization sub(A) = $U^H U$ or sub(A) = LL^H , as computed by p?potrf.

ia,ja (global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix sub(A), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix A.

anorm (global) REAL for single precision flavors,
DOUBLE PRECISION for double precision flavors.
The 1-norm of the symmetric/Hermitian distributed matrix sub(A).

work (local)
REAL for pspocon
DOUBLE PRECISION for pdpocon
COMPLEX for pcpocon
DOUBLE COMPLEX for pzpocon.

The array *work* of dimension (*lwork*) is a workspace array.

lwork (local or global) INTEGER. The dimension of the array *work*.
For real flavors:
lwork must be at least
lwork ≥ 2*LOC_r(*n*+mod(*ia*-1,*mb_a*))+
2*LOC_c(*n*+mod(*ja*-1,*nb_a*))+

max(2, max(*nb_a**ceil(NPROW-1, NPCOL), LOC_c(*n*+mod(*ja*-1,*nb_a*))+ *nb_a**ceil(NPCOL-1, NPROW))).

For complex flavors:
lwork must be at least
lwork ≥ 2*LOC_r(*n*+mod(*ia*-1,*mb_a*))+
max(2, max(*nb_a**max(1,ceil(NPROW-1, NPCOL)), LOC_c(*n*+mod(*ja*-1,*nb_a*))+ *nb_a**max(1,ceil(NPCOL-1, NPROW)))).

iwork (local) INTEGER.
Workspace array, DIMENSION (*liwork*). Used in real flavors only.

liwork (local or global) INTEGER.
The dimension of the array *iwork*; used in real flavors only. Must be at least *liwork* ≥ LOC_r(*n*+mod(*ia*-1,*mb_a*)).

rwork (local) REAL for pcpocon
DOUBLE PRECISION for pzpocon
Workspace array, DIMENSION (*lrwork*). Used in complex flavors only.

lrwork (local or global) INTEGER.
The dimension of the array *rwork*; used in complex flavors only. Must be at least *lrwork* ≥ 2*LOC_c(*n*+mod(*ja*-1,*nb_a*)).

Output Parameters

rcond (global) REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
The reciprocal of the condition number of the distributed matrix sub(*A*).

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

iwork(1) On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).

rwork(1) On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).

info (global) INTEGER. If *info*=0, the execution is successful.

info < 0:

if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.

p?trcon

Estimates the reciprocal of the condition number of a triangular distributed matrix in either 1-norm or infinity-norm.

Syntax

```
call pstrcon ( norm, uplo, diag, n, a, ia, ja, desca,  rcond, work,
    lwork, iwork, liwork, info )
call pdtrcon ( norm, uplo, diag, n, a, ia, ja, desca,  rcond, work,
    lwork, iwork, liwork, info )
call pctrcon ( norm, uplo, diag, n, a, ia, ja, desca,  rcond, work,
    lwork, rwork, lrwork, info )
call pztrcon ( norm, uplo, diag, n, a, ia, ja, desca,  rcond, work,
    lwork, rwork, lrwork, info )
```

Description

This routine estimates the reciprocal of the condition number of a triangular distributed matrix sub(*A*) = *A*(*ia:ia+n-1*, *ja:ja+n-1*), in either the 1 - norm or the infinity-norm.

The norm of sub(*A*) is computed and an estimate is obtained for $\|(\text{sub}(A))^{-1}\|$, then the reciprocal of the condition number is computed as

$$rcond = \frac{1}{\|sub(A)\| \times \|(sub(A))^{-1}\|}$$

Input Parameters

norm (global) CHARACTER*1. Must be '1' or 'O' or 'I'.

Specifies whether the 1-norm condition number or the infinity-norm condition number is required.

If *norm* = '1' or 'O', then the 1-norm is used;

If *norm* = 'I', then the infinity-norm is used.

uplo (global) CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', sub(*A*) is upper triangular.
If *uplo* = 'L', sub(*A*) is lower triangular.

diag (global) CHARACTER*1. Must be 'N' or 'U'.

If *diag* = 'N', sub(*A*) is non-unit triangular.
If *diag* = 'U', sub(*A*) is unit triangular.

n (global) INTEGER. The order of the distributed submatrix sub(*A*) ($n \geq 0$).

a (local)
REAL for pstrcon
DOUBLE PRECISION for pdtrcon
COMPLEX for pctrcon
DOUBLE COMPLEX for pztrcon.

Pointer into the local memory to an array of dimension
a(*lld_a*, LOC_c(*ja*+*n*-1)).

The array *a* contains the local pieces of the triangular distributed matrix sub(*A*).
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of this distributed matrix contains the upper triangular matrix, and its strictly lower triangular part is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of this distributed matrix contains the lower triangular matrix, and its strictly upper triangular part is not referenced.
If *diag* = 'U', the diagonal elements of sub(*A*) are also not referenced and are assumed to be 1.

ia,ja (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local)
REAL for pstrcon
DOUBLE PRECISION for pdtrcon
COMPLEX for pctrcon
DOUBLE COMPLEX for pztrcon.

The array *work* of dimension (*lwork*) is a workspace array.

lwork (local or global) INTEGER. The dimension of the array *work*.

For real flavors:

lwork must be at least

lwork ≥ 2*LOC_r(*n*+mod(*ia*-1,*mb_a*))+
LOC_c(*n*+mod(*ja*-1,*nb_a*))+
max(2, max(*nb_a**max(1, ceil(NPROW-1, NPCOL)),
LOC_c(*n*+mod(*ja*-1,*nb_a*))+
*nb_a**max(1, ceil(NPCOL-1, NPROW))).

For complex flavors:

lwork must be at least

lwork ≥ 2*LOC_r(*n*+mod(*ia*-1,*mb_a*))+
max(2, max(*nb_a**ceil(NPROW-1, NPCOL),
LOC_c(*n*+mod(*ja*-1,*nb_a*))+
*nb_a**ceil(NPCOL-1, NPROW))).

iwork (local) INTEGER.
Workspace array, DIMENSION (*liwork*). Used in real flavors only.

liwork (local or global) INTEGER.
The dimension of the array *iwork*; used in real flavors only. Must be at least *liwork* ≥ LOC_r(*n*+mod(*ia*-1,*mb_a*)).

rwork (local) REAL for pcpocon
DOUBLE PRECISION for pzpocon
Workspace array, DIMENSION (*lrwork*). Used in complex flavors only.

lrwork (local or global) INTEGER.
The dimension of the array *rwork*; used in complex flavors only. Must be at least *lrwork* ≥ LOC_c(*n*+mod(*ja*-1,*nb_a*)).

Output Parameters

rcond (global) REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
The reciprocal of the condition number of the distributed matrix sub(*A*).

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

iwork(1) On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).

rwork(1) — On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).

info — (global) INTEGER. If *info*=0, the execution is successful.

info < 0:

if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.

Refining the Solution and Estimating Its Error

This section describes the ScaLAPACK routines for refining the computed solution of a system of linear equations and estimating the solution error. You can call these routines after factorizing the matrix of the system of equations and computing the solution (see "Routines for Matrix Factorization" and "Routines for Solving Systems of Linear Equations").

p?gerfs

Improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution.

Syntax

```
call psgerfs (trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf,
   ipiv, b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork,
   iwork, liwork, info)
call pdgerfs (trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf,
   ipiv, b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork,
   iwork, liwork, info)
call pcgerfs (trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf,
   ipiv, b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork,
   rwork, lrwork, info)
call pzgerfs (trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf,
   ipiv, b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork,
   rwork, lrwork, info)
```

Description

This routine improves the computed solution to one of the systems of linear equations

sub(A)*sub(X) = sub(B) ,
sub(A)^T*sub(X) = sub(B) , or
sub(A)^T*sub(X) = sub(B)

and provides error bounds and backward error estimates for the solution.

Here sub(A) = A(`ia:ia+n-1`, `ja:ja+n-1`), sub(B) = B(`ib:ib+n-1`, `jb:jb+nrhs-1`) , and sub(X) = X(`ix:ix+n-1`, `jx:jx+nrhs-1`).

Input Parameters

trans (global) `CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`.

Specifies the form of the system of equations:

If *trans* = `'N'`, the system has the form sub(A)*sub(X) = sub(B) (No transpose);

If *trans* = `'T'`, the system has the form sub(A)^T*sub(X) = sub(B) (Transpose);

If *trans* = `'C'`, the system has the form sub(A)^H*sub(X) = sub(B) (Conjugate transpose).

n (global) `INTEGER`. The order of the distributed submatrix sub(A) ($n \geq 0$).

nrhs (global) `INTEGER`. The number of right-hand sides, i.e., the number of columns of the matrices sub(B) and sub(X) ($nrhs \geq 0$).

a, af, b, x (local)
`REAL` for `psgerfs`
`DOUBLE PRECISION` for `pdgerfs`
`COMPLEX` for `pcgerfs`
`DOUBLE COMPLEX` for `pzgerfs`.

Pointers into the local memory to arrays of local dimension `a(`*lld_a*`,` LOC_c`(`*ja+n-1*`))`, `af(`*lld_af*`,` LOC_c`(`*jaf+n-1*`))`, `b(`*lld_b*`,` LOC_c`(`*jb+nrhs-1*`))`, and `x(`*lld_x*`,` LOC_c`(`*jx+nrhs-1*`))`, respectively.

The array *a* contains the local pieces of the distributed matrix sub(A).

The array *af* contains the local pieces of the distributed factors of the matrix sub(A) = $P\ L\ U$ as computed by `p?getrf`.

The array *b* contains the local pieces of the distributed matrix of right hand sides sub(*B*).

On entry, the array *x* contains the local pieces of the distributed solution matrix sub(*X*).

ia,ja (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

iaf,jaf (global) INTEGER. The row and column indices in the global array *AF* indicating the first row and the first column of the submatrix sub(*AF*), respectively.

descaf (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *AF*.

ib,jb (global) INTEGER. The row and column indices in the global array *B* indicating the first row and the first column of the submatrix sub(*B*), respectively.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

ix,jx (global) INTEGER. The row and column indices in the global array *X* indicating the first row and the first column of the submatrix sub(*X*), respectively.

descx (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *X*.

ipiv (local) INTEGER.
Array, dimension $LOC_r(m\_af) + mb\_af$.
This array contains pivoting information as computed by p?getrf. If *ipiv*(i)=j , then the local row i was swapped with the global row j.
This array is tied to the distributed matrix *A*.

work (local)
REAL for psgerfs
DOUBLE PRECISION for pdgerfs
COMPLEX for pcgerfs
DOUBLE COMPLEX for pzgerfs.

The array *work* of dimension (*lwork*) is a workspace array.

lwork (local or global) INTEGER. The dimension of the array *work*.
For real flavors:
lwork must be at least
$lwork \geq 3*LOC_r(n+\text{mod}(ia-1,mb\_a))$

For complex flavors:
lwork must be at least
$lwork \geq 2*LOC_r(n+\text{mod}(ia-1,mb\_a))$

iwork (local) INTEGER.
Workspace array, DIMENSION (*liwork*). Used in real flavors only.

liwork (local or global) INTEGER.
The dimension of the array *iwork*; used in real flavors only. Must be at least $liwork \geq LOC_r(n+\text{mod}(ib-1,mb\_b))$.

rwork (local) REAL for pcgerfs
DOUBLE PRECISION for pzgerfs
Workspace array, DIMENSION (*lrwork*). Used in complex flavors only.

lrwork (local or global) INTEGER.
The dimension of the array *rwork*; used in complex flavors only. Must be at least $lrwork \geq LOC_r(n+\text{mod}(ib-1,mb\_b)))$.

Output Parameters

x On exit, contains the improved solution vectors.

ferr, berr REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, dimension $LOC_c(jb+nrhs-1)$ each.

The array *ferr* contains the estimated forward error bound for each solution vector of sub(*X*).
If XTRUE is the true solution corresponding to sub(*X*), *ferr* is an estimated upper bound for the magnitude of the largest element in (sub(*X*) - XTRUE) divided by the magnitude of the largest element in sub(*X*). The estimate is as reliable as the estimate for rcond, and is almost always a slight overestimate of the true error.
This array is tied to the distributed matrix *X*.

The array *berr* contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of sub(*A*) or sub(*B*) that makes sub(*X*) an exact solution). This array is tied to the distributed matrix *X*.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

iwork(1) On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).

rwork(1) On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).

info (global) INTEGER. If *info*=0, the execution is successful.

info < 0:

if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.

p?porfs

Improves the computed solution to a system of linear equations with symmetric/Hermitian positive definite distributed matrix and provides error bounds and backward error estimates for the solution.

Syntax

```
call psporfs (uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, b,
    ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork,
    liwork, info)
call pdporfs (uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, b,
    ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork,
    liwork, info)
call pcporfs (uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, b,
    ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork,
    lrwork, info)
call pzporfs (uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, b,
    ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork,
    lrwork, info)
```

Description

The routine `p?porfs` improves the computed solution to the system of linear equations

sub(A)*sub(X) = sub(B) ,

where sub(A) = A(`ia:ia+n-1`, `ja:ja+n-1`) is a real symmetric or complex Hermitian positive definite distributed matrix and

sub(B) = B(`ib:ib+n-1`, `jb:jb+nrhs-1`) ,

sub(X) = X(`ix:ix+n-1`, `jx:jx+nrhs-1`)

are right-hand side and solution submatrices, respectively.
This routine also provides error bounds and backward error estimates for the solution.

Input Parameters

uplo (global) `CHARACTER*1`. Must be `'U'` or `'L'`.

Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(A) is stored.

If *uplo* = `'U'`, sub(A) is upper triangular.
If *uplo* = `'L'`, sub(A) is lower triangular.

n (global) `INTEGER`. The order of the distributed matrix sub(A) ($n \geq 0$).

nrhs (global) `INTEGER`. The number of right-hand sides, i.e., the number of columns of the matrices sub(B) and sub(X) ($nrhs \geq 0$).

a, af, b, x (local)
`REAL` for `psporfs`
`DOUBLE PRECISION` for `pdporfs`
`COMPLEX` for `pcporfs`
`DOUBLE COMPLEX` for `pzporfs`.

Pointers into the local memory to arrays of local dimension `a(lld_a,` LOC_c`(ja+n-1))`, `af(lld_af,` LOC_c`(ja+n-1))`, `b(lld_b,` LOC_c`(jb+nrhs-1))`, and `x(lld_x,` LOC_c`(jx+nrhs-1))`, respectively.

The array *a* contains the local pieces of the *n*-by-*n* symmetric/Hermitian distributed matrix sub(A).
If *uplo* = `'U'`, the leading *n*-by-*n* upper triangular part of sub(A) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.
If *uplo* = `'L'`, the leading *n*-by-*n* lower triangular part of sub(A) contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced.

The array *af* contains the factors *L* or *U* from the Cholesky factorization sub(A) = LL^H or sub(A) = U^HU, as computed by p?potrf.

On entry, the array *b* contains the local pieces of the distributed matrix of right hand sides sub(*B*).

On entry, the array *x* contains the local pieces of the solution vectors sub(*X*).

ia,ja (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

iaf,jaf (global) INTEGER. The row and column indices in the global array *AF* indicating the first row and the first column of the submatrix sub(*AF*), respectively.

descaf (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *AF*.

ib,jb (global) INTEGER. The row and column indices in the global array *B* indicating the first row and the first column of the submatrix sub(*B*), respectively.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

ix,jx (global) INTEGER. The row and column indices in the global array *X* indicating the first row and the first column of the submatrix sub(*X*), respectively.

descx (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *X*.

work (local)
REAL for psporfs
DOUBLE PRECISION for pdporfs
COMPLEX for pcporfs
DOUBLE COMPLEX for pzporfs.

The array *work* of dimension (*lwork*) is a workspace array.

lwork (local) INTEGER. The dimension of the array *work*.
For real flavors:
lwork must be at least
lwork ≥ 3*LOC_r(*n*+mod(*ia*-1,*mb_a*))

For complex flavors:
lwork must be at least
lwork ≥ 2*LOC_r(*n*+mod(*ia*-1,*mb_a*))

iwork (local) INTEGER.
Workspace array, DIMENSION (*liwork*). Used in real flavors only.

liwork (local or global) INTEGER.
The dimension of the array *iwork*; used in real flavors only. Must be at least *liwork* ≥ LOC_r(*n*+mod(*ib*-1,*mb_b*)).

rwork (local) REAL for pcporfs
DOUBLE PRECISION for pzporfs
Workspace array, DIMENSION (*lrwork*). Used in complex flavors only.

lrwork (local or global) INTEGER.
The dimension of the array *rwork*; used in complex flavors only. Must be at least *lrwork* ≥ LOC_r(*n*+mod(*ib*-1,*mb_b*))).

Output Parameters

x On exit, contains the improved solution vectors.

ferr, berr REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, dimension LOC_c(*jb*+*nrhs*-1) each.

The array *ferr* contains the estimated forward error bound for each solution vector of sub(*X*).
If XTRUE is the true solution corresponding to sub(*X*), *ferr* is an estimated upper bound for the magnitude of the largest element in (sub(*X*) - XTRUE) divided by the magnitude of the largest element in sub(*X*). The estimate is as reliable as the estimate for rcond, and is almost always a slight overestimate of the true error.
This array is tied to the distributed matrix *X*.

The array *berr* contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of sub(*A*) or sub(*B*) that makes sub(*X*) an exact solution). This array is tied to the distributed matrix *X*.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

iwork(1) On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).

rwork(1) On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).

info (global) INTEGER. If *info*=0, the execution is successful.

info < 0:

if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.

p?trrfs

Provides error bounds and backward error estimates for the solution to a system of linear equations with a distributed triangular coefficient matrix.

Syntax

```
call pstrrfs (uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb,
    descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)
call pdtrrfs (uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb,
    descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)
call pctrrfs (uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb,
    descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)
call pztrrfs (uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb,
    descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)
```

Description

The routine p?trrfs provides error bounds and backward error estimates for the solution to one of the systems of linear equations

$$\text{sub}(A)*\text{sub}(X) = \text{sub}(B)\ ,$$
$$\text{sub}(A)^T*\text{sub}(X) = \text{sub}(B)\ , \text{ or}$$
$$\text{sub}(A)^T*\text{sub}(X) = \text{sub}(B)\ ,$$

where sub(A) = A(*ia:ia+n-1*, *ja:ja+n-1*) is a triangular matrix,
sub(B) = B(*ib:ib+n-1*, *jb:jb+nrhs-1*) , and
sub(X) = X(*ix:ix+n-1*, *jx:jx+nrhs-1*).

The solution matrix X must be computed by `p?trtrs` or some other means before entering this routine. The routine `p?trrfs` does not do iterative refinement because doing so cannot improve the backward error.

Input Parameters

uplo — (global) `CHARACTER*1`. Must be `'U'` or `'L'`.

If *uplo* = `'U'`, sub(A) is upper triangular.
If *uplo* = `'L'`, sub(A) is lower triangular.

trans — (global) `CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`.

Specifies the form of the system of equations:

If *trans* = `'N'`, the system has the form
sub(A)*sub(X) = sub(B) (No transpose);

If *trans* = `'T'`, the system has the form
sub(A)^T*sub(X) = sub(B) (Transpose);

If *trans* = `'C'`, the system has the form
sub(A)^H*sub(X) = sub(B) (Conjugate transpose).

diag — `CHARACTER*1`. Must be `'N'` or `'U'`.

If *diag* = `'N'`, then sub(A) is non-unit triangular.

If *diag* = `'U'`, then sub(A) is unit triangular.

n — (global) `INTEGER`. The order of the distributed matrix sub(A) ($n \ge 0$).

nrhs — (global) `INTEGER`. The number of right-hand sides, i.e., the number of columns of the matrices sub(B) and sub(X) ($nrhs \ge 0$).

a, b, x — (local)
`REAL` for `pstrrfs`
`DOUBLE PRECISION` for `pdtrrfs`
`COMPLEX` for `pctrrfs`
`DOUBLE COMPLEX` for `pztrrfs`.

Pointers into the local memory to arrays of local dimension *a(lld_a, LOC_c(ja+n-1))*, *b(lld_b, LOC_c(jb+nrhs-1))*, and *x(lld_x, LOC_c(jx+nrhs-1))*, respectively.

The array *a* contains the local pieces of the original triangular distributed matrix sub(*A*).
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced.
If *diag* = 'U', the diagonal elements of sub(*A*) are also not referenced and are assumed to be 1.

On entry, the array *b* contains the local pieces of the distributed matrix of right hand sides sub(*B*).

On entry, the array *x* contains the local pieces of the solution vectors sub(*X*).

ia,ja (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

ib,jb (global) INTEGER. The row and column indices in the global array *B* indicating the first row and the first column of the submatrix sub(*B*), respectively.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

ix,jx (global) INTEGER. The row and column indices in the global array *X* indicating the first row and the first column of the submatrix sub(*X*), respectively.

descx (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *X*.

work (local)
REAL for pstrrfs
DOUBLE PRECISION for pdtrrfs
COMPLEX for pctrrfs
DOUBLE COMPLEX for pztrrfs.

The array *work* of dimension (*lwork*) is a workspace array.

lwork (local) INTEGER. The dimension of the array *work*.
For real flavors:
lwork must be at least
$lwork \geq 3*LOC_r(n+mod(ia-1,mb\_a))$

For complex flavors:
lwork must be at least
$lwork \geq 2*LOC_r(n+mod(ia-1,mb\_a))$

iwork (local) INTEGER.
Workspace array, DIMENSION (*liwork*). Used in real flavors only.

liwork (local or global) INTEGER.
The dimension of the array *iwork*; used in real flavors only. Must be at least
$liwork \geq LOC_r(n+mod(ib-1,mb\_b))$.

rwork (local) REAL for pctrrfs
DOUBLE PRECISION for pztrrfs
Workspace array, DIMENSION (*lrwork*). Used in complex flavors only.

lrwork (local or global) INTEGER.
The dimension of the array *rwork*; used in complex flavors only. Must be at least $lrwork \geq LOC_r(n+mod(ib-1,mb\_b)))$.

Output Parameters

ferr, berr REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, dimension $LOC_c(jb+nrhs-1)$ each.

The array *ferr* contains the estimated forward error bound for each solution vector of sub(*X*).
If XTRUE is the true solution corresponding to sub(*X*), *ferr* is an estimated upper bound for the magnitude of the largest element in (sub(*X*) - XTRUE) divided by the magnitude of the largest element in sub(*X*). The estimate is as

reliable as the estimate for `rcond`, and is almost always a slight overestimate of the true error.
This array is tied to the distributed matrix *X*.

The array *berr* contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of sub(*A*) or sub(*B*) that makes sub(*X*) an exact solution). This array is tied to the distributed matrix *X*.

work(1) — On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

iwork(1) — On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).

rwork(1) — On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).

info — (global) INTEGER. If *info*=0, the execution is successful.

info < 0:

if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.

Routines for Matrix Inversion

This sections describes ScaLAPACK routines that compute the inverse of a matrix based on the previously obtained factorization. Note that it is not recommended to solve a system of equations $Ax = b$ by first computing A^{-1} and then forming the matrix-vector product $x = A^{-1}b$.
Call a solver routine instead (see "Routines for Solving Systems of Linear Equations"); this is more efficient and more accurate.

p?getri

Computes the inverse of a LU-factored distributed matrix.

Syntax

```
call psgetri (n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork,
    info)
call pdgetri (n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork,
    info)
call pcgetri (n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork,
    info)
call pzgetri (n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork,
    info)
```

Description

This routine computes the inverse of a general distributed matrix sub(A) = A(`ia:ia+n-1`, `ja:ja+n-1`) using the LU factorization computed by `p?getrf`. This method inverts U and then computes the inverse of sub(A) denoted by $InvA$ by solving the system

$$InvA * L = U^{-1}$$

for $InvA$.

Input Parameters

n (global) `INTEGER`. The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(A) ($n \geq 0$).

a (local)
`REAL` for `psgetri`
`DOUBLE PRECISION` for `pdgetri`
`COMPLEX` for `pcgetri`
`DOUBLE COMPLEX` for `pzgetri`.

Pointer into the local memory to an array of local dimension `a(lld_a,` LOC_c`(ja+n-1))`.

On entry, the array *a* contains the local pieces of the L and U obtained by the factorization sub(A) = $P\ L\ U$ computed by `p?getrf`.

ia, ja (global) `INTEGER`. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local)
`REAL` for `psgetri`
`DOUBLE PRECISION` for `pdgetri`
`COMPLEX` for `pcgetri`
`DOUBLE COMPLEX` for `pzgetri`.

The array *work* of dimension (*lwork*) is a workspace array.

lwork (local) `INTEGER`. The dimension of the array *work*.
lwork must be at least
lwork ≥ LOC_r(*n*+mod(*ia*-1,*mb_a*))**nb_a* .
The array *work* is used to keep at most an entire column block of sub(*A*).

iwork (local) `INTEGER`.
Workspace array used for physically transposing the pivots, `DIMENSION` (*liwork*).

liwork (local or global) `INTEGER`.
The dimension of the array *iwork*.
The minimal value *liwork* of is determined by the following code:

```
If NPROW == NPCOL then
 liwork = LOCc(n_a + mod(ja-1,nb_a))+ nb_a
Else
 liwork = LOCc(n_a + mod(ja-1,nb_a)) +
max(ceil(ceil(LOCr(m_a)/mb_a)/(lcm/NPROW)),nb_a)
End if
```

where `lcm` is the least common multiple of process rows and columns (`NPROW` and `NPCOL`).

Output Parameters

ipiv (local) `INTEGER`.
Array, dimension (LOC_r(*m_a*)+ *mb_a*).
This array contains the pivoting information.
If *ipiv*(i)=j , then the local row i was swapped with the global row j.
This array is tied to the distributed matrix *A*.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

iwork(1) On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance.

info (global) INTEGER. If *info*=0, the execution is successful.

info < 0:
if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.

info > 0:
if *info* = *i*, $U(i,i)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, and division by zero will occur if it is used to solve a system of equations.

p?potri

Computes the inverse of a symmetric/Hermitian positive definite distributed matrix.

Syntax

```
call pspotri (uplo, n, a, ia, ja, desca, info)
call pdpotri (uplo, n, a, ia, ja, desca, info)
call pcpotri (uplo, n, a, ia, ja, desca, info)
call pzpotri (uplo, n, a, ia, ja, desca, info)
```

Description

This routine computes the inverse of a real symmetric or complex Hermitian positive definite distributed matrix sub(A) = A(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) using the Cholesky factorization sub(A) = $U^H U$ or sub(A) = LL^H computed by p?potrf.

Input Parameters

uplo (global) CHARACTER*1. Must be 'U' or 'L'.

Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(A) is stored.

If *uplo* = 'U', upper triangle of sub(*A*) is stored.
If *uplo* = 'L', lower triangle of sub(*A*) is stored.

n (global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(*A*) ($n \geq 0$).

a (local)
REAL for pspotri
DOUBLE PRECISION for pdpotri
COMPLEX for pcpotri
DOUBLE COMPLEX for pzpotri.

Pointer into the local memory to an array of local dimension a(*lld_a*, LOC_c(*ja*+*n*-1)).

On entry, the array *a* contains the local pieces of the triangular factor *U* or *L* from the Cholesky factorization sub(*A*) = $U^H U$ or sub(*A*) = LL^H, as computed by p?potrf.

ia, *ja* (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

a On exit, overwritten by the local pieces of the upper or lower triangle of the (symmetric/Hermitian) inverse of sub(*A*).

info (global) INTEGER. If *info*=0, the execution is successful.

info < 0:
if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.

info > 0:
if *info* = *i*, the (*i*,*i*) element of the factor *U* or *L* is zero, and the inverse could not be computed.

p?trtri

Computes the inverse of a triangular distributed matrix.

Syntax

```
call pstrtri (uplo, diag, n, a, ia, ja, desca, info)
call pdtrtri (uplo, diag, n, a, ia, ja, desca, info)
call pctrtri (uplo, diag, n, a, ia, ja, desca, info)
call pztrtri (uplo, diag, n, a, ia, ja, desca, info)
```

Description

This routine computes the inverse of a real or complex upper or lower triangular distributed matrix sub(A) = A(*ia:ia+n-1*, *ja:ja+n-1*).

Input Parameters

uplo (global) `CHARACTER*1`. Must be `'U'` or `'L'`.

Specifies whether the distributed matrix sub(A) is upper or lower triangular.

If *uplo* = `'U'`, sub(A) is upper triangular.
If *uplo* = `'L'`, sub(A) is lower triangular.

diag `CHARACTER*1`. Must be `'N'` or `'U'`.
Specifies whether or not the distributed matrix sub(A) is unit triangular.

If *diag* = `'N'`, then sub(A) is non-unit triangular.

If *diag* = `'U'`, then sub(A) is unit triangular.

n (global) `INTEGER`. The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(A) ($n \geq 0$).

a (local)
`REAL` for `pstrtri`
`DOUBLE PRECISION` for `pdtrtri`
`COMPLEX` for `pctrtri`
`DOUBLE COMPLEX` for `pztrtri`.

Pointer into the local memory to an array of local dimension `a(`*lld_a*, LOC_c`(`*ja+n-1*`))`.

The array *a* contains the local pieces of the triangular distributed matrix sub(*A*).
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular matrix to be inverted, and the strictly lower triangular part of sub(*A*) is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular matrix, and the strictly upper triangular part of sub(*A*) is not referenced.

ia, ja (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

a On exit, overwritten by the (triangular) inverse of the original matrix.

info (global) INTEGER. If *info*=0, the execution is successful.

info < 0:
if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.

info > 0:
if *info* = *k*, *A*(*ia*+*k*-1, *ja*+*k*-1) is exactly zero. The triangular matrix sub(*A*) is singular and its inverse can not be computed.

Routines for Matrix Equilibration

ScaLAPACK routines described in this section are used to compute scaling factors needed to equilibrate a matrix. Note that these routines do not actually scale the matrices.

p?geequ

Computes row and column scaling factors intended to equilibrate a general rectangular distributed matrix and reduce its condition number.

Syntax

```
call psgeequ (m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
call pdgeequ (m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
call pcgeequ (m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
call pzgeequ (m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
```

Description

This routine computes row and column scalings intended to equilibrate an *m*-by-*n* distributed matrix sub(A) = A(*ia:ia+m-1*, *ja:ja+n-1*) and reduce its condition number. The output array *r* returns the row scale factors and the array *c* the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements b_{ij}=*r*(i)*a_{ij}**c*(j) have absolute value 1.

r(i) and *c*(j) are restricted to be between *SMLNUM* = smallest safe number and *BIGNUM* = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of sub(A) but works well in practice.

Input Parameters

m (global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(A) ($m \geq 0$).

n (global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(A) ($n \geq 0$).

a (local)
REAL for psgeequ
DOUBLE PRECISION for pdgeequ
COMPLEX for pcgeequ
DOUBLE COMPLEX for pzgeequ .

Pointer into the local memory to an array of local dimension a(*lld_a*, LOC_c(*ja+n-1*)).

The array *a* contains the local pieces of the *m*-by-*n* distributed matrix whose equilibration factors are to be computed.

ia,ja (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

r, c (local) REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
Arrays, dimension LOC_r(*m_a*) and LOC_c(*n_a*), respectively.
If *info* = 0, or *info* > *ia+m-1*, the array *r* (*ia:ia+m-1*) contains the row scale factors for sub(*A*). *r* is aligned with the distributed matrix *A*, and replicated across every process column. *r* is tied to the distributed matrix *A*.
If *info* = 0 , the array *c* (*ja:ja+n-1*) contains the column scale factors for sub(*A*). *c* is aligned with the distributed matrix *A*, and replicated down every process row. *c* is tied to the distributed matrix *A*.

rowcnd,colcnd (global) REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
If *info* = 0 or *info* > *ia+m-1*, *rowcnd* contains the ratio of the smallest *r*(i) to the largest *r*(i) ($ia \le i \le ia+m-1$) . If *rowcnd* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *r* (*ia:ia+m-1*).

If *info* = 0, *colcnd* contains the ratio of the smallest *c*(j) to the largest *c*(j) ($ja \le j \le ja+n-1$).
If *colcnd* ≥ 0.1 , it is not worth scaling by *c* (*ja:ja+n-1*).

amax (global) REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
Absolute value of the largest matrix element. If *amax* is very close to overflow or very close to underflow, the matrix should be scaled.

info (global) INTEGER. If *info*=0, the execution is successful.

info < 0:
if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.

`info` > 0:
If `info` = `i` and
`i`≤ `m`, the `i`th row of the distributed matrix sub(*A*) is exactly zero;
`i` > `m`, the (`i-m`)th column of the distributed matrix sub(*A*) is exactly zero.

p?poequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite distributed matrix and reduce its condition number.

Syntax

```
call pspoequ (n, a, ia, ja, desca, sr, sc, scond, amax, info)
call pdpoequ (n, a, ia, ja, desca, sr, sc, scond, amax, info)
call pcpoequ (n, a, ia, ja, desca, sr, sc, scond, amax, info)
call pzpoequ (n, a, ia, ja, desca, sr, sc, scond, amax, info)
```

Description

This routine computes row and column scalings intended to equilibrate a real symmetric or complex Hermitian positive definite distributed matrix sub(*A*) = *A*(`ia:ia+n-1`, `ja:ja+n-1`) and reduce its condition number (with respect to the two-norm). The output arrays `sr` and `sc` return the row and column scale factors

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled distributed matrix *B* with elements $b_{ij}=s(i)*a_{ij}*s(j)$ has ones on the diagonal.

This choice of `sr` and `sc` puts the condition number of *B* within a factor `n` of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

n (global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(*A*) ($n \geq 0$).

a (local)
REAL for pspoequ
DOUBLE PRECISION for pdpoequ
COMPLEX for pcpoequ
DOUBLE COMPLEX for pzpoequ .

Pointer into the local memory to an array of local dimension *a*(*lld_a*, LOC_c(*ja*+*n*-1)).

The array *a* contains the *n*-by-*n* symmetric/Hermitian positive definite distributed matrix sub(*A*) whose scaling factors are to be computed. Only the diagonal elements of sub(*A*) are referenced.

ia,ja (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

sr, sc (local) REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
Arrays, dimension LOC_r(*m_a*) and LOC_c(*n_a*), respectively.
If *info* = 0, the array *sr* (*ia*:*ia*+*n*-1) contains the row scale factors for sub(*A*). *sr* is aligned with the distributed matrix *A*, and replicated across every process column. *sr* is tied to the distributed matrix *A*.
If *info* = 0 , the array *sc* (*ja*:*ja*+*n*-1) contains the column scale factors for sub(*A*). *sc* is aligned with the distributed matrix *A*, and replicated down every process row. *sc* is tied to the distributed matrix *A*.

scond (global) REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
If *info* = 0, *scond* contains the ratio of the smallest *sr*(i) (or *sc*(j)) to the largest *sr*(i) (or *sc*(j)), with
$ia \leq i \leq ia+n-1$ and $ja \leq j \leq ja+n-1$.
If *scond* $\geq$ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *sr* (or *sc*).

amax (global) `REAL` for single precision flavors;
`DOUBLE PRECISION` for double precision flavors.
Absolute value of the largest matrix element. If *amax* is very close to overflow or very close to underflow, the matrix should be scaled.

info (global) `INTEGER`. If *info*=0, the execution is successful.

info < 0:
if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = `-(i*100+j)`; if the *i*th argument is a scalar and had an illegal value, then *info* = `-i`.

info > 0:
If *info* = *k*, the *k*th diagonal entry of sub(*A*) is nonpositive.

Orthogonal Factorizations

This section describes the ScaLAPACK routines for the *QR (RQ)* and *LQ (QL)* factorization of matrices. Routines for the *RZ* factorization as well as for generalized *QR* and *RQ* factorizations are also included. For the mathematical definition of the factorizations, see the respective LAPACK sections or refer to [SLUG].

Table 5-1 lists ScaLAPACK routines that perform orthogonal factorization of matrices.

Table 6-3 Computational Routines for Orthogonal Factorizations

Matrix type, factorization	Factorize without pivoting	Factorize with pivoting	Generate matrix Q	Apply matrix Q
general matrices, QR factorization	p?geqrf	p?geqpf	p?orgqr p?ungqr	p?ormqr p?unmqr
general matrices, RQ factorization	p?gerqf		p?orgrq p?ungrq	p?ormrq p?unmrq
general matrices, LQ factorization	p?gelqf		p?orglq p?unglq	p?ormlq p?unmlq
general matrices, QL factorization	p?geqlf		p?orgql p?ungql	p?ormql p?unmql
trapezoidal matrices, RZ factorization	p?tzrzf			p?ormrz p?unmrz
pair of matrices, generalized QR factorization	p?ggqrf			
pair of matrices, generalized RQ factorization	p?ggrqf			

p?geqrf

Computes the QR factorization of a general m by n matrix.

Syntax

```
call psgeqrf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
call pdgeqrf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
call pcgeqrf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
call pzgeqrf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
```

Description

The routine forms the *QR* factorization of a general *m* by *n* distributed matrix sub(*A*)= *A* (ia:ia+m-1,ja:ja+n-1) as

$$A = Q R$$

Input Parameters

m — (global) INTEGER. The number of rows in the distributed submatrix sub(*A*); ($m \geq 0$).

n — (global) INTEGER. The number of columns in the distributed submatrix sub(*A*); ($n \geq 0$).

a — (local)

REAL for psgeqrf
DOUBLE PRECISION for pdgeqrf
COMPLEX for pcgeqrf
DOUBLE COMPLEX for pzgeqrf.
Pointer into the local memory to an array of local dimension (lld_a, LOCc(ja+n-1)).
Contains the local pieces of the distributed matrix sub(*A*) to be factored.

ia, ja — (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*(ia:ia+m-1,ja:ja+n-1), respectively.

desca — (global and local) INTEGER array, dimension (dlen_). The array descriptor for the distributed matrix *A*

work — (local).
REAL for psgeqrf
DOUBLE PRECISION for pdgeqrf.
COMPLEX for pcgeqrf.
DOUBLE COMPLEX for pzgeqrf
Workspace array of dimension *lwork*.

lwork — (local or global) INTEGER, dimension of *work*, must be at least *lwork* ≥ *nb_a* * (*mp*0+*nq*0+*nb_a*), where

iroff = mod(*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mp*0 = numroc(*m*+*iroff*, *mb_a*, MYROW, *iarow*, NPROW),

*nq*0 = numroc(*n*+*icoff*, *nb_a*, MYCOL, *iacol*, NPCOL),
and numroc, indxg2p are ScaLAPACK tool functions;
MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — The elements on and above the diagonal of sub(A) contain the min(m,n)-by-n upper trapezoidal matrix R (R is upper triangular if $m \ge n$); the elements below the diagonal, with the array *tau*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see *Application Notes* below).

tau — (local)

REAL for psgeqrf
DOUBLE PRECISION for pdgeqrf
COMPLEX for pcgeqrf
DOUBLE COMPLEX for pzgeqrf.
Array, DIMENSION *LOCc*(*ja*+min(*m*,*n*)-1).
Contains the scalar factor *tau* of elementary reflectors. *tau* is tied to the distributed matrix A.

work(1) — On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0, the execution is successful.
< 0, if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix Q is represented as a product of elementary reflectors
$Q = H(ja)\, H(ja+1) \ldots H(ja+k-1)$,

where $k = \min(m,n)$.

Each $H(i)$ has the form

$H(j) = I - tau * v * v'$

where *tau* is a real/complex scalar, and *v* is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$, and *tau* in $tau(ja+i-1)$.

p?geqpf

Computes the QR factorization of a general m by n matrix with pivoting.

Syntax

```
call psgeqpf ( m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info )
call pdgeqpf ( m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info )
call pcgeqpf ( m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info )
call pzgeqpf ( m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info )
```

Description

The routine forms the *QR* factorization with column pivoting of a general *m* by *n* distributed matrix sub(A)= $A(ia:ia+m-1, ja:ja+n-1)$ as

sub$(A)\ P = Q\ R$

Input Parameters

m (global) `INTEGER`. The number of rows in the submatrix sub(A) ($m \geq 0$).

n (global) `INTEGER`. The number of columns in the submatrix sub(A) ($n \geq 0$).

a (local)

`REAL` for `psgeqpf`
`DOUBLE PRECISION` for `pdgeqpf`
`COMPLEX` for `pcgeqpf`
`DOUBLE COMPLEX` for `pzgeqpf`.
Pointer into the local memory to an array of local dimension *(lld_a, LOCc(ja+n-1))*.
Contains the local pieces of the distributed matrix sub(A) to be factored.

ia, *ja* — (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1), respectively.

desca — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

work — (local).
REAL for psgeqpf
DOUBLE PRECISION for pdgeqpf.
COMPLEX for pcgeqpf.
DOUBLE COMPLEX for pzgeqpf
Workspace array of dimension *lwork*.

lwork — (local or global) INTEGER, dimension of *work*, must be at least
For real flavors:
lwork ≥ max (3,*mp*0+*nq*0) + *LOCc* (*ja*+*n*-1) + *nq*0.

For complex flavors:
lwork ≥ max (3,*mp*0+*nq*0) .

Here
iroff = mod(*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mp*0 = numroc (*m*+*iroff*, *mb_a*, MYROW, *iarow*, NPROW),

*nq*0 = numroc (*n*+*icoff*, *nb_a*, MYCOL, *iacol*, NPCOL),

LOCc (*ja*+*n*-1) = numroc(*ja*+*n*-1, *nb_a*, MYCOL,*csrc_a*, NPCOL),
and numroc, indxg2p are ScaLAPACK tool functions;
MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — The elements on and above the diagonal of sub(*A*) contain the min(*m*,*n*)-by-*n* upper trapezoidal matrix *R* (*R* is upper triangular if $m \geq n$); the elements below the diagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below)

ipiv — (local) INTEGER.
Array, DIMENSION *LOCc* (*ja*+*n*-1).

ipiv(*i*) = *k*, the local *i*-th column of sub(*A*)**P* was the global *k*-th column of sub(*A*). *ipiv* is tied to the distributed matrix *A*.

tau — (local)

REAL for psgeqpf
DOUBLE PRECISION for pdgeqpf
COMPLEX for pcgeqpf
DOUBLE COMPLEX for pzgeqpf.
Array, DIMENSION *LOCc*(*ja*+min(*m*,*n*)-1)).
Contains the scalar factor *tau* of elementary reflectors. *tau* is tied to the distributed matrix *A*.

work(1) — On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0, the execution is successful.
< 0, if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors
$Q = H(1)\,H(2)\ldots H(n)$,

Each *H*(*i*) has the form
$H = I - tau * v * v'$

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1:*i*-1) = 0 and *v*(*i*) = 1; *v*(*i*+1:*m*) is stored on exit in *A*(*ia*+*i*-1:*ia*+*m*-1,*ja*+*i*-1).

The matrix *P* is represented in *jpvt* as follows: if *jpvt*(*j*) = *i* then the *j*-th column of *P* is the *i*-th canonical unit vector.

p?orgqr

Generates the orthogonal matrix Q of the QR factorization formed by `p?geqrf`.

Syntax

```
call psorgqr ( m, n, k, a, ia, ja, desca, tau, work, lwork, info )
call pdorgqr ( m, n, k, a, ia, ja, desca, tau, work, lwork, info )
```

Description

The routine generates the whole or part of *m* by *n* real distributed matrix *Q* denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal columns, which is defined as the first *n* columns of a product of *k* elementary reflectors of order *m*

$$Q = H(1)\,H(2)\ldots H(k)$$

as returned by `p?geqrf`.

Input Parameters

m (global) `INTEGER`. The number of rows in the submatrix sub(*Q*) ($m \ge 0$).

n (global) `INTEGER`. The number of columns in the submatrix sub(*Q*) ($m \ge n \ge 0$).

k (global) `INTEGER`. The number of elementary reflectors whose product defines the matrix *Q* ($n \ge k \ge 0$).

a (local)

`REAL` for `psorgqr`
`DOUBLE PRECISION` for `pdorgqr`
Pointer into the local memory to an array of local dimension $(lld\_a, LOCc(ja+n-1))$.The *j*-th column must contain the vector which defines the elementary reflector $H(j)$, $ja \le j \le ja+k-1$, as returned by `p?geqrf` in the *k* columns of its distributed matrix argument $a(ia:*, ja:ja+k-1)$.

ia, ja (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.

desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
tau	(local) REAL for psorgqr DOUBLE PRECISION for pdorgqr Array, DIMENSION *LOCc*(*ja*+*k*-1)). Contains the scalar factor *tau* (*j*) of elementary reflectors *H*(*j*) as returned by p?geqrf. *tau* is tied to the distributed matrix *A*.
work	(local) REAL for psorgqr DOUBLE PRECISION for pdorgqr Workspace array of dimension of *lwork*.
lwork	(local or global) INTEGER, dimension of *work.* Must be at least *lwork* ≥ *nb_a** (*nqa*0 + *mpa*0 + *nb_a*), where *iroffa* = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*), *iarow* = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW), *iacol* = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL), *mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW), *nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL) ; indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo. If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a	Contains the local pieces of the *m*-by-*n* distributed matrix Q.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?ungqr

Generates the complex unitary matrix Q of the QR factorization formed by p?geqrf.

Syntax

```
call pcungqr ( m, n, k, a, ia, ja, desca, tau, work, lwork, info )
call pzungqr ( m, n, k, a, ia, ja, desca, tau, work, lwork, info )
```

Description

The routine generates the whole or part of *m* by *n* complex distributed matrix *Q* denoting *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) 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 p?geqrf.

Input Parameters

m (global) INTEGER. The number of rows in the submatrix sub(*Q*);($m \geq 0$).

n (global) INTEGER. The number of columns in the submatrix sub(*Q*) ($m \geq n \geq 0$).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q* ($n \geq k \geq 0$).

a (local)

COMPLEX for pcungqr
DOUBLE COMPLEX for pzungqr
Pointer into the local memory to an array of dimension (*lld_a*,

	LOCc(*ja*+*n*-1)).The *j*-th column must contain the vector which defines the elementary reflector *H*(*j*), *ja* ≤ *j* ≤ *ja* +*k*-1, as returned by `p?geqrf` in the *k* columns of its distributed matrix argument *a*(*ia*:*,*ja*:*ja*+*k*-1).
ia,ja	(global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
tau	(local) `COMPLEX` for `pcungqr` `DOUBLE COMPLEX` for `pzungqr` Array, `DIMENSION` *LOCc*(*ja*+*k*-1)). Contains the scalar factor *tau* (*j*) of elementary reflectors *H*(*j*) as returned by `p?geqrf`. *tau* is tied to the distributed matrix *A*.
work	(local) `COMPLEX` for `pcungqr` `DOUBLE COMPLEX` for `pzungqr` Workspace array of dimension of *lwork*.
lwork	(local or global) `INTEGER`, dimension of *work*, must be at least *lwork* ≥ *nb_a** (*nqa*0 + *mpa*0 + *nb_a*), where *iroffa* = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*), *iarow* = `indxg2p`(*ia*, *mb_a*, `MYROW`, *rsrc_a*, `NPROW`), *iacol* = `indxg2p`(*ja*, *nb_a*, `MYCOL`, *csrc_a*, `NPCOL`), *mpa*0 = `numroc`(*m*+*iroffa*, *mb_a*, `MYROW`, *iarow*, `NPROW`), *nqa*0 = `numroc`(*n*+*icoffa*, *nb_a*, `MYCOL`, *iacol*, `NPCOL`) `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`. If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a — Contains the local pieces of the *m* by *n* distributed matrix *Q*.

`work(1)` — On exit `work(1)` contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?ormqr

Multiplies a general matrix by the orthogonal matrix Q of the QR factorization formed by `p?geqrf`.

Syntax

```
call psormqr ( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
call pdormqr ( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
```

Description

The routine overwrites the general real *m*-by-*n* distributed matrix sub(C) = C(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	Q sub(C)	sub(C) Q
trans = 'T':	Q^T sub(C)	sub(C) Q^T

where Q is a real orthogonal distributed matrix defined as the product of *k* elementary reflectors

$Q = H(1)\,H(2)\ldots H(k)$

as returned by `p?geqrf`. Q is of order *m* if *side* ='L' and of order *n* if *side* ='R'.

Input Parameters

side (global) CHARACTER
='L': Q or Q^T is applied from the left.
='R': Q or Q^T is applied from the right.

trans (global) CHARACTER
='N', no transpose, Q is applied.
='T', transpose, Q^T is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub(C) ($m \geq 0$).

n (global) INTEGER. The number of columns in the distributed matrix sub(C) ($n \geq 0$).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints:
if *side* ='L', $m \geq k \geq 0$
if *side* ='R', $n \geq k \geq 0$.

a (local)

REAL for psormqr
DOUBLE PRECISION for pdormqr.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*k*-1)).The *j*-th column must contain the vector which defines the elementary reflector $H(j)$, $ja \leq j \leq ja + k - 1$, as returned by p?geqrf in the *k* columns of its distributed matrix argument *a*(*ia*:*,*ja*:*ja*+*k*-1). *a*(*ia*:*,*ja*:*ja*+*k*-1) is modified by the routine but restored on exit.

if *side* ='L', *lld_a* ≥ max (1, *LOCr*(*ia*+*m*-1)

if *side* ='R', *lld_a* ≥ max (1, *LOCr*(*ia*+*n*-1)

ia, ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix A, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix A.

tau (local)
REAL for psormqr
DOUBLE PRECISION for pdormqr

Array, DIMENSION $LOCc(ja+k-1)$.).
Contains the scalar factor *tau* (*j*) of elementary reflectors $H(j)$ as returned by p?geqrf. *tau* is tied to the distributed matrix *A*.

c (local)

REAL for psormqr
DOUBLE PRECISION for pdormqr
Pointer into the local memory to an array of local dimension $(lld\_c, LOCc(jc+n-1))$.

Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic,jc (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)

REAL for psormqr
DOUBLE PRECISION for pdormqr. Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:
if *side* ='L',

$lwork \geq \max((nb\_a*(nb\_a-1))/2, (nqc0 + mpc0)*nb\_a) + nb\_a * nb\_a$

else if *side* ='R',

$lwork \geq \max((nb\_a*(nb\_a-1))/2, (nqc0 + \max(npa0 + \text{numroc}(\text{numroc}(n+icoffc, nb\_a, 0, 0, \text{NPCOL}), nb\_a, 0, 0, lcmq), mpc0))*nb\_a) + nb\_a * nb\_a$

end if

where

$lcmq = lcm / \text{NPCOL}$ with $lcm = \text{ilcm}(\text{NPROW}, \text{NPCOL})$,

$\text{iroffa} = \text{mod}(ia-1, mb\_a)$,

$icoffa = \text{mod}(ja-1, nb\_a)$,

$iarow = \text{indxg2p}(ia, mb\_a, \text{MYROW}, rsrc\_a, \text{NPROW})$,

$npa0 = \text{numroc}(n+iroffa, mb\_a, \text{MYROW}, iarow, \text{NPROW})$,

$iroffc = \text{mod}(ic-1, mb\_c)$,

icoffc = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

if *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — Overwritten by the product Q* sub(C) or Q^T sub (C), or sub(C)* Q^T, or sub(C)* Q .

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?unmqr

Multiplies a complex matrix by the unitary matrix Q of the QR factorization formed by `p?geqrf`.

Syntax

```
call cunmqr ( side,trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
call zunmqr ( side,trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
```

Description

The routine overwrites the general complex *m*-by-*n* distributed matrix sub $(C) = C(ic:ic+m-1,jc:jc+n-1)$ with

	side = 'L'	*side* = 'R'
trans = 'N':	Q sub(C)	sub(C) Q
trans = 'T':	Q^H sub(C)	sub(C) Q^H

where Q is a complex unitary distributed matrix defined as the product of *k* elementary reflectors

$Q = H(1)\ H(2) \dots H(k)$

as returned by `p?geqrf`. Q is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side (global) CHARACTER
= 'L': Q or Q^H is applied from the left.
= 'R': Q or Q^H is applied from the right.

trans (global) CHARACTER
= 'N', no transpose, Q is applied.
= 'C', conjugate transpose, Q^H is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub(C) ($m \geq 0$).

n (global) INTEGER. The number of columns in the distributed matrix sub(C) ($n \geq 0$).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:
if *side* = 'L', $m \geq k \geq 0$
if *side* = 'R', $n \geq k \geq 0$.

a (local)

COMPLEX for pcunmqr
DOUBLE COMPLEX for pzunmqr.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*k*-1)).The *j*-th column must contain the vector which defines the elementary reflector *H*(*j*), $ja \leq j \leq ja + k-1$, as returned by p?geqrf in the *k* columns of its distributed matrix argument *a*(*ia*:*,*ja*:*ja*+*k*-1). *a*(*ia*:*,*ja*:*ja*+*k*-1) is modified by the routine but restored on exit.

if *side* = 'L', *lld_a* ≥ max (1, *LOCr*(*ia*+*m*-1)

if *side* = 'R', *lld_a* ≥ max (1, *LOCr*(*ia*+*n*-1)

ia,ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)
COMPLEX for pcunmqr
DOUBLE COMPLEX for pzunmqr
Array, DIMENSION *LOCc*(*ja*+*k*-1)).
Contains the scalar factor *tau* (*j*) of elementary reflectors *H*(*j*) as returned by p?geqrf. *tau* is tied to the distributed matrix *A*.

c (local)

COMPLEX for pcunmqr
DOUBLE COMPLEX for pzunmqr.
Pointer into the local memory to an array of local dimension (*lld_c*, *LOCc*(*jc*+*n*-1)).

Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic,jc (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)

COMPLEX for pcunmqr
DOUBLE COMPLEX for pzunmqr. Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:
if *side* ='L',

lwork ≥ max ((*nb_a**(*nb_a*-1))/2, (*nqc*0 + *mpc*0)**nb_a*) + *nb_a* * *nb_a*

else if *side* ='R',

lwork ≥ max ((*nb_a** (*nb_a*-1))/2, (*nqc*0 + max
(*npa*0 + numroc (numroc(*n*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a* * *nb_a*

end if

where

lcmq = *lcm* / NPCOL with *lcm* = ilcm (NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p (*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

*npa*0 = numroc(*n*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

if *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c	Overwritten by the product Q^* sub(C) or Q^H sub (C), or sub(C)* Q^H, or sub(C)* Q .
`work(1)`	On exit `work(1)` contains the minimum value of *lwork* required for optimum performance.
info	(global) `INTEGER`. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?gelqf

Computes the LQ factorization of a general rectangular matrix.

Syntax

```
call psgelqf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
call pdgelqf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
call pcgelqf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
call pzgelqf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
```

Description

The routine computes the LQ factorization of a real/complex distributed *m* by *n* matrix sub(A)= A(*ia*:*ia*+*m*-1,*ia*:*ia*+*n*-1) = $L*Q$

Input Parameters

m	(global) `INTEGER`. The number of rows in the submatrix sub(Q);($m \geq 0$).
n	(global) `INTEGER`. The number of columns in the submatrix sub(Q) ($n \geq 0$).
k	(global) `INTEGER`. The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
a	(local)

REAL for psgelqf
DOUBLE PRECISION for pdgelqf
COMPLEX for pcgelqf
DOUBLE COMPLEX for pzgelqf
Pointer into the local memory to an array of local dimension *(lld_a, LOCc(ja+n-1))*.Contains the local pieces of the distributed matrix sub(*A*) to be factored.

ia,ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*((*ia*:*ia*+*m*-1,*ia*:*ia*+*n*-1), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local)

REAL for psgelqf
DOUBLE PRECISION for pdgelqf
COMPLEX for pcgelqf
DOUBLE COMPLEX for pzgelqf
Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least *lwork* ≥ *mb_a** (*mp*0 + *nq*0 + *mb_a*), where

iroff = mod(*ia*-1, *mb_a*),

icoff = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mp*0 = numroc (*m*+*iroff*, *mb_a*, MYROW, *iarow*, NPROW),

*nq*0 = numroc (*n*+*icoff*, *nb_a*, MYCOL, *iacol*, NPCOL)

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — The elements on and below the diagonal of sub(*A*) contain the *m* by min(*m*,*n*) lower trapezoidal matrix *L* (*L* is lower trapezoidal if $m \le n$); the elements above the diagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below)

tau — (local)

`REAL` for `psgelqf`
`DOUBLE PRECISION` for `pdgelqf`
`COMPLEX` for `pcgelqf`
`DOUBLE COMPLEX` for `pzgelqf`
Array, `DIMENSION` *LOCr*(*ia*+min(*m*,*n*)-1)).
Contains the scalar factors of elementary reflectors. *tau* is tied to the distributed matrix *A*.

work(1) — On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) `INTEGER`.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

$Q = H(ia+k-1)\ H(ia+k-2) \dots H(ia)$,

where $k = \min(m,n)$

Each *H*(*i*) has the form

$H(i) = I - tau * v * v'$

where *tau* is a real/complex scalar, and *v* is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:n)$ is stored on exit in $A(ia+i-1:ia+i-1, ja+n-1)$, and *tau* in *tau* $(ia+i-1)$.

p?orglq

Generates the real orthogonal matrix Q of the LQ factorization formed by `p?gelqf`.

Syntax

```
call psorglq ( m, n, k, a, ia, ja, desca, tau, work, lwork, info )
call pdorglq ( m, n, k, a, ia, ja, desca, tau, work, lwork, info )
```

Description

The routine generates the whole or part of *m* by *n* real distributed matrix *Q* denoting $A(ia{:}ia+m-1, ja{:}ja+n-1)$ 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 `p?gelqf`.

Input Parameters

m — (global) `INTEGER`. The number of rows in the submatrix sub(*Q*); ($m \ge 0$).

n — (global) `INTEGER`. The number of columns in the submatrix sub(*Q*) ($n \ge m \ge 0$).

k — (global) `INTEGER`. The number of elementary reflectors whose product defines the matrix *Q* ($m \ge k \ge 0$).

a — (local)

`REAL` for `psorglq`
`DOUBLE PRECISION` for `pdorglq`
Pointer into the local memory to an array of local dimension *(lld_a, LOCc(ja+n-1))*.On entry, the *i*-th row must contain the vector which defines the elementary reflector *H*(*i*), $ia \le i \le ia+k-1$, as returned by `p?gelqf` in the *k* rows of its distributed matrix argument $A(ia{:}ia+k-1, ja{:}*)$.

ia, ja — (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix $A((ia{:}ia+m-1, ja{:}ja+n-1)$, respectively.

desca	(global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
work	(local) `REAL` for `psorglq` `DOUBLE PRECISION` for `pdorglq` Workspace array of dimension of *lwork*.
lwork	(local or global) `INTEGER`, dimension of *work*, must be at least *lwork* ≥ *mb_a** (*mpa*0 + *nqa*0 + *mb_a*), where *iroffa* = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*), *iarow* = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW), *iacol* = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL), *mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW), *nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL) `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW, MYCOL, NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`. If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a	Contains the local pieces of the *m*-by-*n* distributed matrix *Q* to be factored.
tau	(local) `REAL` for `psorglq` `DOUBLE PRECISION` for `pdorglq` Array, `DIMENSION` *LOCr*(*ia*+*k*-1).). Contains the scalar factors *tau* of elementary reflectors *H*(*i*). *tau* is tied to the distributed matrix *A*.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info (global) `INTEGER`.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?unglq

Generates the unitary matrix Q of the LQ factorization formed by `p?gelqf`.

Syntax

```
call pcunglq ( m, n, k, a, ia, ja, desca, tau, work, lwork, info )
call pzunglq ( m, n, k, a, ia, ja, desca, tau, work, lwork, info )
```

Description

The routine generates the whole or part of *m* by *n* complex distributed matrix *Q* denoting *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) 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 `p?gelqf`.

Input Parameters

m (global) `INTEGER`. The number of rows in the submatrix sub(*Q*);($m \geq 0$).

n (global) `INTEGER`. The number of columns in the submatrix sub(*Q*) ($n \geq m \geq 0$).

k (global) `INTEGER`. The number of elementary reflectors whose product defines the matrix *Q* ($m \geq k \geq 0$).

a (local)

`COMPLEX` for `pcunglq`
`DOUBLE COMPLEX` for `pzunglq`
Pointer into the local memory to an array of local dimension (*lld_a*,

LOCc(*ja*+*n*-1)).On entry, the *i*-th row must contain the vector which defines the elementary reflector *H*(*i*), *ia* ≤ *i* ≤ *ia*+*k*-1, as returned by `p?gelqf` in the *k* rows of its distributed matrix argument *A*(*ia*:*ia*+*k* -1,*ja*:*).

ia, ja — (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1), respectively.

desca — (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau — (local)

`COMPLEX` for `pcunglq`
`DOUBLE COMPLEX` for `pzunglq`
Array, `DIMENSION` *LOCr*(*ia*+*k*-1)).
Contains the scalar factors *tau* of elementary reflectors *H*(*i*). *tau* is tied to the distributed matrix *A*.

work — (local)

`COMPLEX` for `pcunglq`
`DOUBLE COMPLEX` for `pzunglq`
Workspace array of dimension of *lwork*.

lwork — (local or global) `INTEGER`, dimension of *work*, must be at least *lwork* ≥ *mb_a** (*mpa*0 + *nqa*0 + *mb_a*), where

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW, MYCOL, NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a — Contains the local pieces of the *m*-by-*n* distributed matrix *Q* to be factored.

work(1) — On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?ormlq

Multiplies a general matrix by the orthogonal matrix Q of the LQ factorization formed by p?gelqf.

Syntax

```
call psormlq ( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc,
    work, lwork, info )
call pdormlq ( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc,
    work, lwork, info )
```

Description

The routine overwrites the general real *m*-by-*n* distributed matrix
sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side = 'L'	*side* = 'R'
trans = 'N':	*Q* sub(*C*)	sub(*C*) *Q*
trans = 'T':	Q^T sub(*C*)	sub(*C*) Q^T

where *Q* is a real orthogonal distributed matrix defined as the product of *k* elementary reflectors

$Q = H(k)...H(2)\, H(1)$

as returned by p?gelqf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side (global) CHARACTER
='L': Q or Q^T is applied from the left.
='R': Q or Q^T is applied from the right.

trans (global) CHARACTER
='N', no transpose, Q is applied.
='T', transpose, Q^T is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub(C) ($m \geq 0$).

n (global) INTEGER. The number of columns in the distributed matrix sub(C) ($n \geq 0$).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints:
if *side* ='L', $m \geq k \geq 0$
if *side* ='R', $n \geq k \geq 0$.

a (local)

REAL for psormlq
DOUBLE PRECISION for pdormlq.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*m*-1)), if *side* ='L' and (*lld_a*, *LOCc*(*ja*+*n*-1)), if *side* ='R'.The *i*-th row must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia + k - 1$, as returned by p?gelqf in the *k* rows of its distributed matrix argument *a*(*ia*:*ia*+*k*-1,*ja*:*).
a(*ia*:*ia*+*k*-1,*ja*:*) is modified by the routine but restored on exit.

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix A, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix A.

tau (local)

REAL for psormlq
DOUBLE PRECISION for pdormlq
Array, DIMENSION *LOCc*(*ja*+*k*-1)).
Contains the scalar factor *tau* (*i*) of elementary reflectors $H(i)$ as returned by p?gelqf. *tau* is tied to the distributed matrix A.

c (local)

REAL for psormlq
DOUBLE PRECISION for pdormlq
Pointer into the local memory to an array of local dimension *(lld_c, LOCc(jc+n-1))*.
Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, jc (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)

REAL for psormlq
DOUBLE PRECISION for pdormlq. Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of the array *work*; must be at least:
if *side* ='L',

lwork ≥ max ((*mb_a**(*mb_a*-1))/2, (*mpc*0 + max *mqa*0)+ numroc (numroc (*m* + *iroffc*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0)) * *mb_a*) + *mb_a***mb_a*

else if *side* ='R',

lwork ≥ max ((*mb_a** (*mb_a*-1))/2, (*mpc*0 + nqc0) **mb_a* + *mb_a***mb_a*

end if

where

lcmp = *lcm* / NPROW with *lcm* = ilcm (NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iacol = indxg2p (*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mqa*0 = *numroc*(*m*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

if *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — Overwritten by the product *Q** sub(*C*) or *Q*' sub (*C*), or sub(*C*)* *Q*', or sub(*C*)* *Q*

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?unmlq

Multiplies a general matrix by the unitary matrix Q of the LQ factorization formed by `p?gelqf`.

Syntax

```
call pcunmlq ( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
call pzunmlq ( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
```

Description

The routine overwrites the general complex *m*-by-*n* distributed matrix sub $(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	side ='L'	*side* ='R'
trans = 'N':	Q sub(C)	sub(C) Q
trans = 'T':	Q^H sub(C)	sub(C) Q^H

where Q is a complex unitary distributed matrix defined as the product of *k* elementary reflectors

$Q = H(k)' \ldots H(2)' H(1)'$

as returned by `p?gelqf`. Q is of order *m* if *side* ='L' and of order *n* if *side* ='R'.

Input Parameters

side (global) CHARACTER
='L': Q or Q^H is applied from the left.
='R': Q or Q^H is applied from the right.

trans (global) CHARACTER
='N', no transpose, Q is applied.
='C', conjugate transpose, Q^H is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub(C) ($m \geq 0$).

n (global) INTEGER. The number of columns in the distributed matrix sub(C) ($n \geq 0$).

k (global) `INTEGER`. The number of elementary reflectors whose product defines the matrix Q. Constraints:
if *side* = 'L', $m \geq k \geq 0$
if *side* = 'R', $n \geq k \geq 0$.

a (local)

`COMPLEX` for `pcunmlq`
`DOUBLE COMPLEX` for `pzunmlq`.
Pointer into the local memory to an array of dimension
(*lld_a*, *LOCc*(*ja*+*m*-1)), if *side* = 'L', and
(*lld_a*, *LOCc*(*ja*+*n*-1)), if *side* = 'R',
where *lld_a* ≥ max (1, *LOCr* (*ia*+*k*-1)).The *i*-th column must contain the vector which defines the elementary reflector $H(i)$, *ia* ≤ *i* ≤ *ia* +*k*-1, as returned by `p?gelqf` in the *k* rows of its distributed matrix argument *a*(*ia*:*ia*+*k*-1,*ja*:*).
a(*ia*:*ia*+*k*-1,*ja*:*) is modified by the routine but restored on exit.

ia,ja (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix A, respectively.

desca (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix A.

tau (local)

`COMPLEX` for `pcunmlq`
`DOUBLE COMPLEX` for `pzunmlq`
Array, `DIMENSION` *LOCc*(*ia*+*k*-1)).
Contains the scalar factor *tau* (*i*) of elementary reflectors $H(i)$ as returned by `p?gelqf`. *tau* is tied to the distributed matrix A.

c (local)

`COMPLEX` for `pcunmlq`
`DOUBLE COMPLEX` for `pzunmlq`.
Pointer into the local memory to an array of local dimension (*lld_c*, *LOCc*(*jc*+*n*-1)).

Contains the local pieces of the distributed matrix sub(C) to be factored.

ic,jc (global) `INTEGER`. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix C, respectively.

descc (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix C.

work (local)

COMPLEX for pcunmlq
DOUBLE COMPLEX for pzunmlq. Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of the array *work*; must be at least:
if *side* ='L',

lwork ≥ max ((*mb_a**(*mb_a*-1))/2, (*mpc*0 + max *mqa*0)+ numroc (numroc (*m* + *iroffc*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0)) * *mb_a*) + *mb_a***mb_a*

else if *side* ='R',

lwork ≥ max ((*mb_a** (*mb_a*-1))/2, (*mpc*0 + nqc0) **mb_a* + *mb_a***mb_a*

end if

where

lcmp = *lcm* / NPROW with *lcm* = ilcm (NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iacol = indxg2p (*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mqa*0 = *numroc*(*m* + *icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

if *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c	Overwritten by the product *Q** sub(*C*) or *Q'* sub (*C*), or sub(*C*)* *Q'*, or sub(*C*)* *Q*
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?geqlf

Computes the QL factorization of a general matrix.

Syntax

```
call psgeqlf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
call pdgeqlf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
call pcgeqlf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
call pzgeqlf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
```

Description

The routine forms the *QL* factorization of a real/complex distributed *m*-by-*n* matrix sub *(A)* = *A* (*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) = *Q* * *L*.

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix sub(*Q*)*;*(*m* ≥ 0).
n	(global) INTEGER. The number of columns in the submatrix sub(*Q*) (*n* ≥ 0).
a	(local) REAL for psgeqlf DOUBLE PRECISION for pdgeqlf COMPLEX for pcgeqlf DOUBLE COMPLEX for pzgeqlf

Pointer into the local memory to an array of local dimension (*lld_a*, *LOCc*(*ja*+*n*-1)).Contains the local pieces of the distributed matrix sub(*A*) to be factored.

ia, *ja* — (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*((*ia*:*ia*+*m*-1,*ia*:*ia*+*n*-1), respectively.

desca — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

work — (local)

REAL for psgeqlf
DOUBLE PRECISION for pdgeqlf
COMPLEX for pcgeqlf
DOUBLE COMPLEX for pzgeqlf
Workspace array of dimension of *lwork*.

lwork — (local or global) INTEGER, dimension of *work*, must be at least *lwork* ≥ *nb_a** (*mp*0 + *nq*0 + *nb_a*), where

iroff = mod(*ia*-1, *mb_a*),

icoff = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mp*0 = numroc (*m*+*iroff*, *mb_a*, MYROW, *iarow*, NPROW),

*nq*0 = numroc (*n*+*icoff*, *nb_a*, MYCOL, *iacol*, NPCOL)

numroc and indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — On exit, if *m* ≥ *n*, the lower triangle of the distributed submatrix *A*(*ia*+*m*-*n*:*ia*+*m*-1, *ja*:*ja*+*n*-1) contains the *n*-by-*n* lower triangular matrix *L*; if *m* ≤ *n*, the elements on and below the (*n*-*m*)-th superdiagonal contain the *m* by *n* lower trapezoidal matrix *L*; the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below)

tau (local)

`REAL` for `psgeqlf`
`DOUBLE PRECISION` for `pdgeqlf`
`COMPLEX` for `pcgeqlf`
`DOUBLE COMPLEX` for `pzgeqlf`
Array, `DIMENSION` *LOCc*(*ja*+*n*-1)).
Contains the scalar factors of elementary reflectors. *tau* is tied to the distributed matrix *A*.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info (global) `INTEGER`.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*ja*+*k*-1)... *H*(*ja*+1) *H*(*ja*),

where *k* = min(*m*,*n*)

Each *H*(*i*) has the form

H(*i*) = *I* - *tau* * *v* * *v*'

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(*m*-*k*+*i*+1:*m*) = 0 and *v*(*m*-*k*+*i*) = 1; *v*(*m*-*k*+*i*-1) is stored on exit in *A*(*ia*+*ia*+*m*-*k*+*i*-2, *ja*+*n*-*k*+*i*-1), and *tau* in *tau* (*ja*+*n*-*k*+*i*-1).

p?orgql

Generates the orthogonal matrix Q of the QL factorization formed by `p?geqlf`.

Syntax

```
call psorgql ( m, n, k, a, ia, ja, desca, tau, work, lwork, info )
call pdorgql ( m, n, k, a, ia, ja, desca, tau, work, lwork, info )
```

Description

The routine generates the whole or part of *m* by *n* real distributed matrix *Q* denoting *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) with orthonormal rows, which is defined as the first *m* rows of a product of *k* elementary reflectors of order *n*

$Q = H(k)\dots H(2)\ H(1)$

as returned by `p?gelqf`.

Input Parameters

m (global) `INTEGER`. The number of rows in the submatrix sub(*Q*);($m \ge 0$).

n (global) `INTEGER`. The number of columns in the submatrix sub(*Q*) ($m \ge n \ge 0$).

k (global) `INTEGER`. The number of elementary reflectors whose product defines the matrix *Q* ($n \ge k \ge 0$).

a (local)

`REAL` for `psorgql`
`DOUBLE PRECISION` for `pdorgql`
Pointer into the local memory to an array of local dimension (*lld_a*, *LOCc*(*ja*+*n*-1)).On entry, the *j*-th column must contain the vector which defines the elementary reflector *H*(*j*), $ja + n - k \le j \le ja+n-1$, as returned by `p?geqlf` in the *k* columns of its distributed matrix argument *A*(*ia*:**ja*+*n*-*k*:*ja*+*n*-1).

ia, *ja* (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*((*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)

REAL for psorgql
DOUBLE PRECISION for pdorgql
Array, DIMENSION *LOCc*(*ja*+*n*-1)).
Contains the scalar factors *tau*(*j*) of elementary reflectors *H*(*j*). *tau* is tied to the distributed matrix *A*.

work (local)

REAL for psorgql
DOUBLE PRECISION for pdorgql
Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least *lwork* ≥ *nb_a** (*nqa*0 + *mpa*0 + *nb_a*), where

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a Contains the local pieces of the *m*-by-*n* distributed matrix *Q* to be factored.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?ungql

Generates the unitary matrix Q of the QL factorization formed by p?geqlf.

Syntax

```
call pcungql ( m, n, k, a, ia, ja, desca, tau, work, lwork, info )
call pzungql ( m, n, k, a, ia, ja, desca, tau, work, lwork, info )
```

Description

The routine generates the whole or part of *m* by *n* complex distributed matrix *Q* denoting *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) with orthonormal rows, which is defined as the first *n* columns of a product of *k* elementary reflectors of order *m*

$Q = H(k)... H(2) H(1)$

as returned by p?geqlf.

Input Parameters

m (global) INTEGER. The number of rows in the submatrix sub(*Q*) ($m \ge 0$).

n (global) INTEGER. The number of columns in the submatrix sub(*Q*) ($m \ge n \ge 0$).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q* ($n \ge k \ge 0$).

a (local)

COMPLEX for pcungql
DOUBLE COMPLEX for pzungql
Pointer into the local memory to an array of local dimension *(lld_a, LOCc(ja+n-1))*.On entry, the *j*-th column must contain the vector which

defines the elementary reflector $H(j)$, $ja+n-k \le j \le ja+n-1$, as returned by `p?geqlf` in the k columns of its distributed matrix argument $A(ia:*, ja+n-k: ja+n-1)$.

ia,ja (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix $A(ia:ia+m-1,ja:ja+n-1)$, respectively.

desca (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)

`COMPLEX` for `pcungql`
`DOUBLE COMPLEX` for `pzungql`
Array, `DIMENSION` $LOCr(ia+n-1)$).
Contains the scalar factors *tau* (*j*) of elementary reflectors $H(j)$. *tau* is tied to the distributed matrix *A*.

work (local)

`COMPLEX` for `pcungql`
`DOUBLE COMPLEX` for `pzungql`
Workspace array of dimension of *lwork*.

lwork (local or global) `INTEGER`, dimension of *work*, must be at least $lwork \ge nb\_a*(nqa0 + mpa0 + nb\_a)$, where

$iroffa$ = mod(ia-1, mb_a),

$icoffa$ = mod(ja-1, nb_a),

$iarow$ = indxg2p(ia, mb_a, MYROW, $rsrc\_a$, NPROW),

$iacol$ = indxg2p(ja, nb_a, MYCOL, $csrc\_a$, NPCOL),

$mpa0$ = numroc($m+iroffa$, mb_a, MYROW, $iarow$, NPROW),

$nqa0$ = numroc($n+icoffa$, nb_a, MYCOL, $iacol$, NPCOL)

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW, MYCOL, NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a	Contains the local pieces of the *m*-by-*n* distributed matrix *Q* to be factored.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?ormql

Multiplies a general matrix by the orthogonal matrix Q of the QL factorization formed by p?geqlf.

Syntax

```
call psormql ( side, trans, m, n, k, a, ia, ja, desca, tau, c,ic, jc,
    descc, work, lwork, info )
call pdormql ( side, trans, m, n, k, a, ia, ja, desca, tau, c,ic, jc,
    descc, work, lwork, info )
```

Description

The routine overwrites the general real *m*-by-*n* distributed matrix sub(*C*) = *C* (*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side = 'L'	*side* = 'R'
trans = 'N':	Q sub(C)	sub(C) Q
trans = 'T':	Q^T sub(C)	sub(C) Q^T

where *Q* is a real orthogonal distributed matrix defined as the product of *k* elementary reflectors

$Q = H(k)' \dots H(2)' H(1)'$

as returned by p?geqlf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side (global) `CHARACTER`
='`L`': Q or Q^T is applied from the left.
='`R`': Q or Q^T is applied from the right.

trans (global) `CHARACTER`
='`N`', no transpose, Q is applied.
='`T`', transpose, Q^T is applied.

m (global) `INTEGER`. The number of rows in the distributed matrix sub(C) ($m \geq 0$).

n (global) `INTEGER`. The number of columns in the distributed matrix sub(C) ($n \geq 0$).

k (global) `INTEGER`. The number of elementary reflectors whose product defines the matrix Q. Constraints:
if *side* ='`L`', $m \geq k \geq 0$
if *side* ='`R`', $n \geq k \geq 0$.

a (local)

`REAL` for `psormql`
`DOUBLE PRECISION` for `pdormql`.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*k*-1)).The *j*-th column must contain the vector which defines the elementary reflector $H(j)$, $ja \leq j \leq ja + k - 1$, as returned by `p?gelqf` in the *k* columns of its distributed matrix argument *a*(*ia*:*,*ja*:*ja*+*k*-1). *a*(*ia*:*,*ja*:*ja*+*k*-1) is modified by the routine but restored on exit.

if *side* ='`L`', *lld_a* ≥ `max` (1, *LOCr*(*ia*+*m*-1)),

if *side* ='`R`', *lld_a* ≥ `max` (1, *LOCr*(*ia*+*n*-1)).

ia, ja (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix A, respectively.

desca (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix A.

tau (local)

	REAL for psormql DOUBLE PRECISION for pdormql. Array, DIMENSION *LOCc*(*ja*+*n*-1)). Contains the scalar factor *tau* (*j*) of elementary reflectors *H*(*j*) as returned by p?geqlf . *tau* is tied to the distributed matrix *A*.
c	(local) REAL for psormql DOUBLE PRECISION for pdormql. Pointer into the local memory to an array of local dimension (*lld_c*, *LOCc*(*jc*+*n*-1)). Contains the local pieces of the distributed matrix sub(*C*) to be factored.
ic,jc	(global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.
descc	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.
work	(local) REAL for psormql. DOUBLE PRECISION for pdormql. Workspace array of dimension of *lwork*.
lwork	(local or global) INTEGER, dimension of *work*, must be at least: if *side* ='L', *lwork* ≥ max ((*nb_a** (*nb_a*-1))/2, (*nqc*0 + mpc0) **nb_a* + *nb_a***nb_a* else if *side* ='R', *lwork* ≥max ((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max *npa*0)+ numroc (numroc (*n* + *icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0)) * *nb_a*) + *nb_a***nb_a* end if where *lcmp* = *lcm* / NPCOL with *lcm* = ilcm (NPROW, NPCOL), iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*), *iarow* = indxg2p (*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW), *npa*0 = *numroc*(*n* + *iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

if *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — Overwritten by the product *Q** sub(*C*) or *Q'* sub (*C*), or sub(*C*)* *Q'*, or sub(*C*)* *Q*

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?unmql

Multiplies a general matrix by the unitary matrix Q of the QL factorization formed by `p?geqlf`.

Syntax

```
call pcunmql ( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
call pzunmql ( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
```

Description

The routine overwrites the general complex *m*-by-*n* distributed matrix sub(*C*) = *C* (*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	Q sub(C)	sub(C) Q
trans = 'C':	Q^H sub(C)	sub(C) Q^H

where *Q* is a complex unitary distributed matrix defined as the product of *k* elementary reflectors

$Q = H(k)' \ldots H(2)' H(1)'$

as returned by `p?geqlf`. *Q* is of order *m* if *side* ='L' and of order *n* if *side* ='R'.

Input Parameters

side — (global) CHARACTER
='L': Q or Q^H is applied from the left.
='R': Q or Q^H is applied from the right.

trans — (global) CHARACTER
='N', no transpose, Q is applied.
='C', conjugate transpose, Q^H is applied.

m — (global) INTEGER. The number of rows in the distributed matrix sub(*C*) ($m \geq 0$).

n — (global) INTEGER. The number of columns in the distributed matrix sub(*C*) ($n \geq 0$).

k (global) `INTEGER`. The number of elementary reflectors whose product defines the matrix Q. Constraints:
if *side* = 'L', $m \geq k \geq 0$
if *side* = 'R', $n \geq k \geq 0$.

a (local)

`COMPLEX` for `pcunmql`
`DOUBLE COMPLEX` for `pzunmql`.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*k*-1)).The *j*-th column must contain the vector which defines the elementary reflector $H(j)$, $ja \leq j \leq ja + k-1$, as returned by `p?geqlf` in the *k* columns of its distributed matrix argument *a*(*ia*:*,*ja*:*ja*+*k*-1).
a(*ia*:*,*ja*:*ja*+*k*-1) is modified by the routine but restored on exit.

if *side* = 'L', $lld\_a \geq \max(1, LOCr(ia+m-1))$,

if *side* = 'R', $lld\_a \geq \max(1, LOCr(ia+n-1))$.

ia,ja (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)

`COMPLEX` for `pcunmql`
`DOUBLE COMPLEX` for `pzunmql`
Array, `DIMENSION` *LOCc*(*ia*+*n*-1)).
Contains the scalar factor *tau* (*j*) of elementary reflectors *H*(*j*) as returned by `p?geqlf`. *tau* is tied to the distributed matrix *A*.

c (local)

`COMPLEX` for `pcunmql`
`DOUBLE COMPLEX` for `pzunmql`.
Pointer into the local memory to an array of local dimension (*lld_c*, *LOCc*(*jc*+*n*-1)).

Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic,jc (global) `INTEGER`. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)

COMPLEX for pcunmql
DOUBLE COMPLEX for pzunmql. Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:
if *side* ='L',

lwork ≥ max ((*nb_a** (*nb_a*-1))/2, (*nqc*0 + mpc0) **nb_a* + *nb_a***nb_a*

else if *side* ='R',

lwork ≥ max ((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max *npa*0)+ numroc (numroc (*n* + *icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0)) * *nb_a*) + *nb_a***nb_a*

end if

where

lcmp = *lcm* / NPCOL with *lcm* = ilcm (NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p (*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

*npa*0 = *numroc* (*n* + *iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

if *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c	Overwritten by the product *Q** sub(*C*) or *Q'* sub (*C*), or sub(*C*)* *Q'*, or sub(*C*)* *Q*
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?gerqf

Computes the RQ factorization of a general rectangular matrix.

Syntax

```
call psgerqf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
call pdgerqf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
call pcgerqf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
call pzgerqf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
```

Description

The routine forms the *QR* factorization of a general *m* by *n* distributed matrix sub(*A*)= *A* (*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) as

$$A = R\,Q$$

Input Parameters

m	(global) INTEGER. The number of rows in the distributed submatrix sub(*A*); ($m \geq 0$).
n	(global) INTEGER. The number of columns in the distributed submatrix sub(*A*);($n \geq 0$).
a	(local)

REAL for psgeqrf
DOUBLE PRECISION for pdgeqrf
COMPLEX for pcgeqrf
DOUBLE COMPLEX for pzgeqrf.
Pointer into the local memory to an array of local dimension *(lld_a, LOCc(ja+n-1))*.
Contains the local pieces of the distributed matrix sub(*A*) to be factored.

ia, ja — (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*(*ia:ia+m-1*,*ja:ja+n-1*), respectively.

desca — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*

work — (local).
REAL for psgeqrf
DOUBLE PRECISION for pdgeqrf.
COMPLEX for pcgeqrf.
DOUBLE COMPLEX for pzgeqrf
Workspace array of dimension *lwork*.

lwork — (local or global) INTEGER, dimension of *work*, must be at least *lwork* ≥ *mb_a* * (*mp*0+*nq*0+*mb_a*), where

iroff = mod(*ia*-1, *mb_a*),

icoff = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mp*0 = numroc (*m*+*iroff*, *mb_a*, MYROW, *iarow*, NPROW),

*nq*0 = numroc (*n*+*icoff*, *nb_a*, MYCOL, *iacol*, NPCOL) and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — On exit, if $m \le n$, the upper triangle of A(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) contains the *m* by *m* upper triangular matrix *R*; if $m \ge n$, the elements on and above the (*m* - *n*)-th subdiagonal contain the *m* by *n* upper trapezoidal matrix *R*; the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below)

tau — (local)

REAL for psgeqrf
DOUBLE PRECISION for pdgeqrf
COMPLEX for pcgeqrf
DOUBLE COMPLEX for pzgeqrf.
Array, DIMENSION LOC_r(*ia*+*m*-1).
Contains the scalar factor *tau* of elementary reflectors. *tau* is tied to the distributed matrix *A*.

work(1) — On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0, the execution is successful.
< 0, if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors
$Q = H(ia)\, H(ia+1) \ldots H(ia+k-1)$,

where $k = \min(m,n)$.

Each $H(i)$ has the form

$H(i) = I - tau * v * v'$

where *tau* is a real/complex scalar, and *v* is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$; $v(1:n-k+i-1)$/conjg $(v(1:n-k+i-1))$ is stored on exit in $A(ia+m-k+i-1, ja:ja+n-k+i-2)$, and *tau* in $tau(ia+m-k+i-1)$.

p?orgrq

Generates the orthogonal matrix Q of the RQ factorization formed by `p?gerqf`.

Syntax

```
call psorgrq ( m, n, k, a, ia, ja, desca, tau, work, lwork, info )
call pdorgrq ( m, n, k, a, ia, ja, desca, tau, work, lwork, info )
```

Description

The routine generates the whole or part of *m* by *n* real distributed matrix *Q* denoting *A*(*ia:ia+m-1,ja:ja+n-1*) with orthonormal columns, which is defined as the last *m* rows of a product of *k* elementary reflectors of order *m*

Q= H(1) H(2)...H(k)

as returned by `p?gerqf`.

Input Parameters

m (global) `INTEGER`. The number of rows in the submatrix sub(*Q*);($m \geq 0$).

n (global) `INTEGER`. The number of columns in the submatrix sub(*Q*) ($n \geq m \geq 0$).

k (global) `INTEGER`. The number of elementary reflectors whose product defines the matrix *Q* ($m \geq k \geq 0$).

a (local)

`REAL` for `psorgrq`
`DOUBLE PRECISION` for `pdorgrq`
Pointer into the local memory to an array of local dimension (*lld_a*, *LOCc*(*ja+n-1*)).The *i*-th column must contain the vector which defines the elementary reflector *H*(*i*), $ja \leq j \leq ja+k-1$, as returned by `p?geqrf` in the *k* columns of its distributed matrix argument *a*(*ia*:*,*ja*:*ja+k*-1).

ia, ja (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*(*ia:ia+m-1,ja:ja+n-1*), respectively.

desca (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)

`REAL` for `psorgrq`
`DOUBLE PRECISION` for `pdorgrq`
Array, `DIMENSION` *LOCc*(*ja*+*k*-1)).
Contains the scalar factor *tau* (*i*) of elementary reflectors *H*(*i*) as returned by `p?gerqf`. *tau* is tied to the distributed matrix *A*.

work (local)

`REAL` for `psorgrq`
`DOUBLE PRECISION` for `pdorgrq`
Workspace array of dimension of *lwork*.

lwork (local or global) `INTEGER`, dimension of *work*, must be at least *lwork* ≥ *mb_a* * (*mpa*0 + *nqa*0 + *mb_a*), where

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a Contains the local pieces of the *m*-by-*n* distributed matrix Q.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?ungrq

Generates the unitary matrix Q of the RQ factorization formed by p?gerqf.

Syntax

```
call pcungrq ( m, n, k, a, ia, ja, desca, tau, work, lwork, info )
call pzungrq ( m, n, k, a, ia, ja, desca, tau, work, lwork, info )
```

Description

The routine generates the *m* by *n* complex distributed matrix *Q* denoting *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) with orthonormal rows, which is defined as the last *m* rows of a product of *k* elementary reflectors of order *n*

$Q = H(1)' H(2)' ... H(k)'$

as returned by p?gerqf.

Input Parameters

m (global) INTEGER. The number of rows in the submatrix sub(*Q*);($m \geq 0$).

n (global) INTEGER. The number of columns in the submatrix sub(*Q*) ($n \geq m \geq 0$).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q* ($m \geq k \geq 0$).

a (local)

COMPLEX for pcungrq
DOUBLE COMPLEX for pzungrq
Pointer into the local memory to an array of dimension (*lld_a*,

$LOCc$(*ja*+*n*-1)).The *i*-th row must contain the vector which defines the elementary reflector $H(i)$, *ia*+*m*-*k* ≤ *i* ≤ *ia* +*m*-1, as returned by `p?gerqf` in the *k* rows of its distributed matrix argument *a*(*ia*+*m*-*k*:*ia*+*m*-1, *ja*:*).

ia,ja (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)

`COMPLEX` for `pcungrq`
`DOUBLE COMPLEX` for `pzungrq`
Array, `DIMENSION` $LOCr$(*ia*+*m*-1)).
Contains the scalar factor *tau* (*i*) of elementary reflectors $H(i)$ as returned by `p?gerqf`. *tau* is tied to the distributed matrix *A*.

work (local)

`COMPLEX` for `pcungrq`
`DOUBLE COMPLEX` for `pzungrq`
Workspace array of dimension of *lwork*.

lwork (local or global) `INTEGER`, dimension of *work*, must be at least *lwork* ≥ *mb_a** (*mpa*0 +*nqa*0+*mb_a*), where

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW, MYCOL, NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a Contains the local pieces of the *m* by *n* distributed matrix Q.

work(1) On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?ormrq

Multiplies a general matrix by the orthogonal matrix Q of the RQ factorization formed by p?gerqf.

Syntax

```
call psormrq ( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
call pdormrq ( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
```

Description

The routine overwrites the general real *m*-by-*n* distributed matrix sub(C) = C(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side = 'L'	*side* = 'R'
trans = 'N':	Q sub(C)	sub(C) Q
trans = 'T':	Q^T sub(C)	sub(C) Q^T

where Q is a real orthogonal distributed matrix defined as the product of *k* elementary reflectors

$Q = H(1)\ H(2) \ldots H(k)$

as returned by p?geqrf. Q is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side
(global) CHARACTER
='L': Q or Q^T is applied from the left.
='R': Q or Q^T is applied from the right.

trans
(global) CHARACTER
='N', no transpose, Q is applied.
='T', transpose, Q^T is applied.

m
(global) INTEGER. The number of rows in the distributed matrix sub(C) ($m \geq 0$).

n
(global) INTEGER. The number of columns in the distributed matrix sub(C) ($n \geq 0$).

k
(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints:
if *side* ='L', $m \geq k \geq 0$
if *side* ='R', $n \geq k \geq 0$.

a
(local)

REAL for psormqr
DOUBLE PRECISION for pdormqr.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*m*-1)) if *side* ='L', and (*lld_a*, *LOCc*(*ja*+*n*-1)) if *side* ='R'.The *i*-th row must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gerqf in the k rows of its distributed matrix argument $a(ia:ia+k-1,ja:*)$. $a(ia:ia+k-1,ja:*)$ is modified by the routine but restored on exit.

ia,ja
(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix A, respectively.

desca
(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix A.

tau
(local)

REAL for psormqr
DOUBLE PRECISION for pdormqr
Array, DIMENSION $LOC_c(ja+k-1)$).
Contains the scalar factor *tau* (i) of elementary reflectors $H(i)$ as returned by p?gerqf. *tau* is tied to the distributed matrix A.

c
(local)

REAL for psormrq
DOUBLE PRECISION for pdormrq
Pointer into the local memory to an array of local dimension *(lld_c, LOCc(jc+n-1))*.

Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic,jc (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)

REAL for psormrq
DOUBLE PRECISION for pdormrq. Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:
if *side* = 'L',

lwork ≥ max ((*mb_a** (*mb_a*-1))/2, (*mpc*0 + max (*mqa*0 + numroc (numroc(*n*+*iroffc*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))**mb_a*) + *mb_a* * *mb_a*

else if *side* = 'R',

lwork ≥ max ((*mb_a**(*mb_a*-1))/2, (*mpc*0 + *nqc*0)**mb_a*) + *mb_a* **mb_a*

end if
where

lcmp = *lcm* / NPROW with *lcm* = ilcm (NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iacol = indxg2p (*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = `numroc`(*n*+*icoffc*, *nb_c*, `MYCOL`, *iccol*, `NPCOL`),

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

if *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

c — Overwritten by the product Q* sub(C) or Q' sub (C), or sub(C)* Q', or sub(C)* Q

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) `INTEGER`.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?unmrq

Multiplies a general matrix by the unitary matrix Q of the RQ factorization formed by `p?gerqf`.

Syntax

```
call pcunmrq ( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
call pzunmrq ( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
```

Description

The routine overwrites the general complex *m*-by-*n* distributed matrix sub (C) = C(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	Q sub(C)	sub(C) Q
trans = 'C':	Q^H sub(C)	sub(C) Q^H

where Q is a complex unitary distributed matrix defined as the product of *k* elementary reflectors

$Q = H(1)' H(2)' \ldots H(k)'$

as returned by `p?gerqf`. Q is of order *m* if *side* ='L' and of order *n* if *side* ='R'.

Input Parameters

side (global) CHARACTER
='L': Q or Q^H is applied from the left.
='R': Q or Q^H is applied from the right.

trans (global) CHARACTER
='N', no transpose, Q is applied.
='C', conjugate transpose, Q^H is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub(C) ($m \geq 0$).

n (global) INTEGER. The number of columns in the distributed matrix sub(C) ($n \geq 0$).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:
if *side* ='L', $m \geq k \geq 0$
if *side* ='R', $n \geq k \geq 0$.

a (local)

COMPLEX for pcunmrq
DOUBLE COMPLEX for pzunmrq.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*m*-1)) if *side* ='L', and (*lld_a*, *LOCc*(*ja*+*n*-1)) if *side* ='R'.The *i*-th row must contain the vector which defines the elementary reflector *H*(*i*), *ia* ≤ *i* ≤ *ia* +*k*-1, as returned by p?gerqf in the *k* rows of its distributed matrix argument *a*(*ia*:*ia* +*k*-1,*ja**). *a*(*ia*:*ia* +*k*-1,*ja**) is modified by the routine but restored on exit.

ia,ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)

COMPLEX for pcunmrq
DOUBLE COMPLEX for pzunmrq
Array, DIMENSION *LOCc*(*ja*+*k*-1)).
Contains the scalar factor *tau* (*i*) of elementary reflectors *H*(*i*) as returned by p?gerqf. *tau* is tied to the distributed matrix *A*.

c (local)

COMPLEX for pcunmrq
DOUBLE COMPLEX for pzunmrq.
Pointer into the local memory to an array of local dimension (*lld_c*, *LOCc*(*jc*+*n*-1)).

Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic,jc (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)

COMPLEX for pcunmrq
DOUBLE COMPLEX for pzunmrq. Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:
if *side* ='L',

lwork ≥ max ((*mb_a** (*mb_a*-1))/2, (*mpc*0 + max (*mqa*0 + numroc (numroc(*n*+*iroffc*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))**mb_a*) + *mb_a* * *mb_a*

else if *side* ='R',

lwork ≥ max ((*mb_a**(*mb_a*-1))/2, (*mpc*0 + *nqc*0)**mb_a*) + *mb_a* * *mb_a*

end if
where

lcmp = *lcm* / NPROW with *lcm* = ilcm (NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iacol = indxg2p (*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mqa*0 = numroc(*m*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

if *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — Overwritten by the product *Q** sub(*C*) or *Q'* sub (*C*), or sub(*C*)* *Q'*, or sub(*C*)* *Q*

`work(1)` — On exit `work(1)` contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?tzrzf

Reduces the upper trapezoidal matrix A to upper triangular form.

Syntax

```
call pstzrzf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
call pdtzrzf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
call pctzrzf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
call pztzrzf ( m, n, a, ia, ja, desca, tau, work, lwork, info )
```

Description

This routine reduces the *m*-by-*n* ($m \le n$) real/complex upper trapezoidal matrix sub(*A*)=(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) to upper triangular form by means of orthogonal/unitary transformations. The upper trapezoidal matrix *A* is factored as

$$A = (R\ 0) * Z,$$

where *Z* is an *n*-by-*n* orthogonal/unitary matrix and R is an *m*-by-*m* upper triangular matrix.

Input Parameters

m — (global) INTEGER. The number of rows in the submatrix sub(*A*);($m \ge 0$).

n — (global) INTEGER. The number of columns in the submatrix sub(*A*) ($n \ge 0$).

a (local)

REAL for pstzrzf
DOUBLE PRECISION for pdtzrzf.
COMPLEX for pctzrzf.
DOUBLE COMPLEX for pztzrzf.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)).Contains the local pieces of the *m* by *n* distributed matrix sub (*A*) to be factored.

ia,ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local)

REAL for pstzrzf
DOUBLE PRECISION for pdtzrzf.
COMPLEX for pctzrzf.
DOUBLE COMPLEX for pztzrzf.
Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least *lwork* ≥ *mb_a** (*mp*0 +*nq*0+*mb_a*), where

iroff = mod(*ia*-1, *mb_a*),

icoff = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mp*0 = numroc (*m*+*iroff*, *mb_a*, MYROW, *iarow*, NPROW),

*nq*0 = numroc (*n*+*icoff*, *nb_a*, MYCOL, *iacol*, NPCOL)

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — On exit, the leading *m*-by-*m* upper triangular part of sub(*A*) contains the upper triangular matrix *R*, and elements *m*+1 to *n* of the first *m* rows of sub (*A*), with the array *tau*, represent the orthogonal/unitary matrix *Z* as a product of *m* elementary reflectors.

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

tau — (local)

REAL for pstzrzf
DOUBLE PRECISION for pdtzrzf.
COMPLEX for pctzrzf.
DOUBLE COMPLEX for pztzrzf.
Array, DIMENSION *LOCr*(*ia*+*m*-1)).
Contains the scalar factor of elementary reflectors. *tau* is tied to the distributed matrix *A*.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The factorization is obtained by the Householder's method. The *k*-th transformation matrix, *Z*(*k*), which is or whose conjugate transpose is used to introduce zeros into the (*m* - *k* + 1)-th row of sub(*A*), is given in the form

$$Z(k) = \begin{bmatrix} i & 0 \\ 0 & T(k) \end{bmatrix}$$

where

$T(k) = i - tau*u(k)*u(k)'$,

$$u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

tau is a scalar and $Z(k)$ is an $(n - m)$ element vector. *tau* and $Z(k)$ are chosen to annihilate the elements of the k-th row of sub(A). The scalar *tau* is returned in the k-th element of *tau* and the vector $u(k)$ in the k-th row of sub(A), such that the elements of $Z(k)$ are in $a(k, m + 1), \ldots, a(k, n)$. The elements of R are returned in the upper triangular part of sub(A). Z is given by

$$Z = Z(1) * Z(2) * \ldots * Z(m).$$

p?ormrz

Multiplies a general matrix by the orthogonal matrix from a reduction to upper triangular form formed by `p?tzrzf`.

Syntax

```
call psormrz ( side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
call pdormrz ( side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
```

Description

The routine overwrites the general real m-by-n distributed matrix sub(C) = $C(ic:ic+m-1, jc:jc+n-1)$ with

	side = 'L'	*side* = 'R'
trans = 'N':	Q sub(C)	sub(C) Q
trans = 'T':	Q^T sub(C)	sub(C) Q^T

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(1)\, H(2) \ldots H(k)$$

as returned by `p?tzrzf`. Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

side — (global) CHARACTER
= 'L': Q or Q^T is applied from the left.
= 'R': Q or Q^T is applied from the right.

trans (global) `CHARACTER`
='N', no transpose, Q is applied.
='T', transpose, Q^T is applied.

m (global) `INTEGER`. The number of rows in the distributed matrix sub(C) ($m \geq 0$).

n (global) `INTEGER`. The number of columns in the distributed matrix sub(C) ($n \geq 0$).

k (global) `INTEGER`. The number of elementary reflectors whose product defines the matrix Q. Constraints:
if *side* ='L', $m \geq k \geq 0$
if *side* ='R', $n \geq k \geq 0$.

l (global)

The columns of the distributed submatrix sub(A) containing the meaningful part of the Householder reflectors.
if *side* ='L', $m \geq l \geq 0$
if *side* ='R', $n \geq l \geq 0$.

a (local)

`REAL` for `psormrz`
`DOUBLE PRECISION` for `pdormrz`.
Pointer into the local memory to an array of dimension (*lld_a*, $LOCc(ja+m-1)$) if *side* ='L', and (*lld_a*, $LOCc(ja+n-1)$) if *side* ='R', where *lld_a* $\geq$ max (1,$LOCr$ ($ia+k-1$).The *i*-th row must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by `p?tzrzf` in the *k* rows of its distributed matrix argument $a(ia:ia+k-1,ja:*)$. $a(ia:ia+k-1,ja:*)$ is modified by the routine but restored on exit.

ia,ja (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix A, respectively.

desca (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix A.

tau (local)

REAL for psormrz
DOUBLE PRECISION for pdormrz
Array, DIMENSION *LOCc*(*ia*+*k*-1)).
Contains the scalar factor *tau* (*i*) of elementary reflectors *H*(*i*) as returned by p?tzrzf. *tau* is tied to the distributed matrix *A*.

c (local)

REAL for psormrz
DOUBLE PRECISION for pdormrz
Pointer into the local memory to an array of local dimension
(*lld_c*, *LOCc*(*jc*+*n*-1)).
Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic,jc (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)

REAL for psormrz
DOUBLE PRECISION for pdormrz. Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:
if *side* ='L',

lwork ≥ max ((*mb_a** (*mb_a*-1))/2, (*mpc*0 + max (*mqa*0 + numroc (numroc(*n*+*iroffc*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))**mb_a*) + *mb_a* * *mb_a*

else if *side* ='R',

lwork ≥ max ((*mb_a**(*mb_a*-1))/2, (*mpc*0 + *nqc*0)**mb_a*) + *mb_a* **mb_a*

end if
where

lcmp = *lcm* / NPROW with *lcm* = ilcm (NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),

iacol = indxg2p (*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

if *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — Overwritten by the product *Q** sub(*C*) or *Q*' sub (*C*), or sub(*C*)**Q'*, or sub(*C*)**Q*

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?unmrz

Multiplies a general matrix by the unitary transformation matrix from a reduction to upper triangular form determined by `p?tzrzf`.

Syntax

```
call pcunmrz ( side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
call pzunmrz ( side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
```

Description

The routine overwrites the general complex *m*-by-*n* distributed matrix sub(C) = C(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side = 'L'	*side* = 'R'
trans = 'N':	Q sub(C)	sub(C) Q
trans = 'C':	Q^H sub(C)	sub(C) Q^H

where Q is a complex unitary distributed matrix defined as the product of *k* elementary reflectors

$Q = H(1)' H(2)' \ldots H(k)'$

as returned by `pctzrzf/pztzrzf`. Q is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side — (global) `CHARACTER`
= 'L': Q or Q^H is applied from the left.
= 'R': Q or Q^H is applied from the right.

trans — (global) `CHARACTER`
= 'N', no transpose, Q is applied.
= 'C', conjugate transpose, Q^H is applied.

m — (global) `INTEGER`. The number of rows in the distributed matrix sub(C) ($m \geq 0$).

n — (global) `INTEGER`. The number of columns in the distributed matrix sub(C) ($n \geq 0$).

k (global) `INTEGER`. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:
if *side* = 'L', $m \geq k \geq 0$
if *side* = 'R', $n \geq k \geq 0$.

a (local)

`COMPLEX` for `pcunmrz`
`DOUBLE COMPLEX` for `pzunmrz`.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*m*-1)) if *side* = 'L', and (*lld_a*, *LOCc*(*ja*+*n*-1)) if *side* = 'R', where *lld_a* ≥ max (1, *LOCr* (*ja*+*k*-1)The *i*-th row must contain the vector which defines the elementary reflector *H*(*i*), *ia* ≤ *i* ≤ *ia* +*k*-1, as returned by `p?gerqf` in the *k* rows of its distributed matrix argument *a*(*ia*:*ia* +*k*-1,*ja**).
a(*ia*:*ia* +*k*-1,*ja**) is modified by the routine but restored on exit.

ia,ja (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)

`COMPLEX` for `pcunmrz`
`DOUBLE COMPLEX` for `pzunmrz`
Array, `DIMENSION` *LOCc*(*ia*+*k*-1)).
Contains the scalar factor *tau* (*i*) of elementary reflectors *H*(*i*) as returned by `p?gerqf`. *tau* is tied to the distributed matrix *A*.

c (local)

`COMPLEX` for `pcunmrz`
`DOUBLE COMPLEX` for `pzunmrz`.
Pointer into the local memory to an array of local dimension (*lld_c*, *LOCc*(*jc*+*n*-1)).
Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic,jc (global) `INTEGER`. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)

COMPLEX for pcunmrz
DOUBLE COMPLEX for pzunmrz. Workspace array of dimension *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:
if *side* ='L',

lwork ≥ max ((*mb_a** (*mb_a*-1))/2, (*mpc*0 + max (*mqa*0 + numroc (numroc(*n*+*iroffc*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))**mb_a*) + *mb_a* * *mb_a*

else if *side* ='R',

lwork ≥ max ((*mb_a**(*mb_a*-1))/2, (*mpc*0 + *nqc*0)**mb_a*) + *mb_a* * *mb_a*

end if

where

lcmp = *lcm* / NPROW with *lcm* = ilcm (NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iacol = indxg2p (*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mqa*0 = numroc(*m*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = *mod*(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

if *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — Overwritten by the product *Q** sub(*C*) or *Q'* sub (*C*), or sub(*C*)**Q'*, or sub(*C*)**Q*

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?ggqrf

Computes the generalized QR factorization.

Syntax

```
call psggqrf (n, m, p, a, ia, ja, desca, taua, b, ib, jb, descb, taub,
    work, lwork, info)
call pdggqrf (n, m, p, a, ia, ja, desca, taua, b, ib, jb, descb, taub,
    work, lwork, info)
call pcggqrf (n, m, p, a, ia, ja, desca, taua, b, ib, jb, descb, taub,
    work, lwork, info)
call pzggqrf (n, m, p, a, ia, ja, desca, taua, b, ib, jb, descb, taub,
    work, lwork, info)
```

Description

The routine forms the generalized *QR* factorization of an *n*-by-*m* matrix

$$\text{sub}(A) = A\,(ia:ia+n-1,\ ja:ja+m-1)$$

and an *n*-by-*p* matrix

$$\text{sub}(B) = B\,(ib:ib+n-1,\ jb:jb+p-1):$$

as

$$\text{sub}(A) = Q\,R, \quad \text{sub}(B) = Q\,T\,Z,$$

where *Q* is an *n*-by-*n* orthogonal/unitary matrix, *Z* is a *p*-by-*p* orthogonal/unitary matrix, and *R* and *T* assume one of the forms:

if $n \geq m$

$$R = \begin{pmatrix} R_{11} \\ 0 \end{pmatrix} \begin{matrix} m \\ n-m \end{matrix}$$

$$m$$

or if $n < m$

$$R = \begin{pmatrix} R_{11} & R_{12} \end{pmatrix} n$$

$$n \quad m-n$$

where R_{11} is upper triangular, and

$$T = \begin{pmatrix} 0 & T_{12} \end{pmatrix} n, \quad \text{if } n \leq p,$$

$$p-n \quad n$$

or $$T = \begin{pmatrix} T_{11} \\ T_{21} \end{pmatrix} \begin{pmatrix} n-p \\ p \end{pmatrix}, \quad \text{if } n > p$$

$$p$$

where T_{12} or T_{21} is an upper triangular matrix.

In particular, if sub(*B*) is square and nonsingular, the *GQR* factorization of sub(*A*) and sub(*B*) implicitly gives the *QR* factorization of inv (sub(*B*))* sub (*A*):

inv (sub(*B*)) * sub(*A*) = $Z^H (T^{-1} R)$

Input Parameters

n (global) INTEGER. The number of rows in the distributed matrices sub (*A*) and sub(*B*) ($n \geq 0$).

m (global) INTEGER. The number of columns in the distributed matrix sub(*A*) ($m \geq 0$).

p INTEGER. The number of columns in the distributed matrix sub(*B*) ($p \geq 0$).

a (local)
REAL for psggqrf
DOUBLE PRECISION for pdggqrf
COMPLEX for pcggqrf
DOUBLE COMPLEX for pzggqrf.
Pointer into the local memory to an array of dimension *(lld_a, LOCc(ja+m-1))*.Contains the local pieces of the *n*-by-*m* matrix sub(*A)* to be factored.

ia,ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

b (local)
REAL for psggqrf
DOUBLE PRECISION for pdggqrf
COMPLEX for pcggqrf
DOUBLE COMPLEX for pzggqrf.
Pointer into the local memory to an array of dimension *(lld_b, LOCc(jb+p-1))*.Contains the local pieces of the *n*-by-*p* matrix sub(*B)* to be factored.

ib,jb (global) INTEGER. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

work (local)
REAL for psggqrf
DOUBLE PRECISION for pdggqrf
COMPLEX for pcggqrf
DOUBLE COMPLEX for pzggqrf.Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least *lwork* ≥ max (*nb_a* * (*npa*0 +*mqa*0+*nb_a*), max((*nb_a**(*nb_a*-1))/2, (*pqb*0 + *npb*0)**nb_a*) + *nb_a* * *nb_a*,
mb_b * (*npb*0 + *pqb*0 + *mb_b*)), *where*

iroffa = mod(*ia-1*, *mb_a)*,

icoffa = mod(*ja-1, nb_a)*,

iarow = indxg2p*(ia, mb_a*, MYROW, *rsrc_a*, NPROW*)*,

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

npa0 = numroc (*n*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

mqa0 = numroc (*m*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)

iroffb = mod(*ib*-1, *mb_b*),

icoffb = mod(*jb*-1, *nb_b*),

ibrow = indxg2p(*ib*, *mb_b*, MYROW, *rsrc_b*, NPROW),

ibcol = indxg2p(*jb*, *nb_b*, MYCOL, *csrc_b*, NPCOL),

npb0 = numroc (*n*+*iroffa*, *mb_b*, MYROW, *ibrow*, NPROW),

pqb0 = numroc (*m*+*icoffb*, *nb_b*, MYCOL, *ibcol*, NPCOL)

and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — On exit, the elements on and above the diagonal of sub (*A*) contain the min(*n*,*m*)-by-*m* upper trapezoidal matrix *R* (*R* is upper triangular if $n \geq m$); the elements below the diagonal, with the array *taua*, represent the orthogonal/unitary matrix *Q* as a product of min(*n*,*m*) elementary reflectors. (See *Application Notes* below).

taua, taub — (local)
REAL for psggqrf
DOUBLE PRECISION for pdggqrf
COMPLEX for pcggqrf
DOUBLE COMPLEX for pzggqrf.
Arrays, DIMENSION *LOCc*(*ja*+min(*n*,*m*)-1)for *taua* and *LOCr*(*ib*+*n*-1) for *taub*.
The array *taua* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q*.*taua* is tied to the distributed matrix *A*.(See *Application Notes* below).

The array *taub* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Z*.*taub* is tied to the distributed matrix *B*.(See *Application Notes* below).

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

$Q = H(ja)\, H(ja+1) \ldots H(ja+k-1)$,

where $k = \min(n,m)$.

Each $H(i)$ has the form

$H(i) = i - taua * v * v'$

where *taua* is a real/complex scalar, and *v* is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:n)$ is stored on exit in $A(ia+i:ia+n-1, ja+i-1)$, and *taua* in $taua(ja+i-1)$.To form *Q* explicitly, use ScaLAPACK subroutine p?orgqr/p?ungqr. To use *Q* to update another matrix, use ScaLAPACK subroutine p?ormqr/p?unmqr.

The matrix *Z* is represented as a product of elementary reflectors

$Z = H(ib)\, H(ib+1) \ldots H(ib+k-1)$,
where $k = \min(n,p)$.

Each $H(i)$ has the form

$H(i) = i - taub * v * v'$

where *taub* is a real/complex scalar, and *v* is a real/complex vector with $v(p-k+i+1:p) = 0$ and $v(p-k+i) = 1$; $v(1:p-k+i-1)$ is stored on exit in $B(ib+n-k+i-1, jb:jb+p-k+i-2)$, and *taub* in $taub(ib+n-k+i-1)$.To form *Z* explicitly, use ScaLAPACK subroutine p?orgrq/p?ungrq.To use *Z* to update another matrix, use ScaLAPACK subroutine p?ormrq/p?unmrq.

p?ggrqf

Computes the generalized RQ factorization.

Syntax

```
call psggrqf (m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub,
    work, lwork, info)
call pdggrqf (m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub,
    work, lwork, info)
call pcggrqf (m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub,
    work, lwork, info)
call pzggrqf (m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub,
    work, lwork, info)
```

Description

The routine forms the generalized *RQ* factorization of an m-by-n matrix sub(A)=(ia:ia+m-1, ja:ja+n-1) and a p-by-n matrix sub(B)=(ib:ib+p-1, ja:ja+n-1):

$$\text{sub}(A) = R\,Q, \quad \text{sub}(B) = Z\,T\,Q,$$

where Q is an n-by-n orthogonal/unitary matrix, Z is a p-by-p orthogonal/unitary matrix, and R and T assume one of the forms:

$$R = m\ \underset{\quad n-m \qquad m}{\begin{pmatrix} 0 & R_{12} \end{pmatrix}}, \quad \text{if } m \le n,$$

or

$$R = \underset{n}{\begin{pmatrix} R_{11} \\ R_{12} \end{pmatrix}} \begin{matrix} m-n \\ n \end{matrix}, \quad \text{if } m > n$$

where R_{11} or R_{21} is upper triangular, and

$$T = \underset{n}{\begin{pmatrix} T_{11} \\ 0 \end{pmatrix}} \begin{matrix} n \\ p-n \end{matrix}, \quad \text{if } p \ge n$$

or

$$T = p \begin{pmatrix} T_{11} & T_{12} \end{pmatrix} \quad p, \text{ if } p < n,$$
$$\qquad\qquad p \qquad n-p$$

where T_{11} is upper triangular.

In particular, if sub(*B*) is square and nonsingular, the *GRQ* factorization of sub(*A*) and sub(*B*) implicitly gives the *RQ* factorization of sub (*A*)*inv(sub(*B*)):

sub(*A*)*inv(sub(*B*)) = (*R**inv(*T*))**Z'*

where inv(sub(*B*)) denotes the inverse of the matrix sub(*B*), and *Z'* denotes the transpose of matrix *Z*.

Input Parameters

m — (global) `INTEGER`. The number of rows in the distributed matrices sub (*A*) ($m \geq 0$).

p — `INTEGER`. The number of rows in the distributed matrix sub(*B*) ($p \geq 0$).

n — (global) `INTEGER`. The number of columns in the distributed matrices sub(*A*) and sub(*B*) ($n \geq 0$).

a — (local)
`REAL` for `psggrqf`
`DOUBLE PRECISION` for `pdggrqf`
`COMPLEX` for `pcggrqf`
`DOUBLE COMPLEX` for `pzggrqf`.
Pointer into the local memory to an array of dimension *(lld_a, LOCc(ja+n-1))*.Contains the local pieces of the *m*-by-*n* distributed matrix sub(*A*) to be factored.

ia, ja — (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca — (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

b — (local)
`REAL` for `psggrqf`
`DOUBLE PRECISION` for `pdggrqf`
`COMPLEX` for `pcggrqf`
`DOUBLE COMPLEX` for `pzggrqf`.

Pointer into the local memory to an array of dimension *(lld_b, LOCc(jb+n-1))*.Contains the local pieces of the *p*-by-*n* matrix sub(*B)* to be factored.

ib,jb (global) INTEGER. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

work (local)
REAL for psggrqf
DOUBLE PRECISION for pdggrqf
COMPLEX for pcggrqf
DOUBLE COMPLEX for pzggrqf.Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least *lwork* ≥ max (*mb_a* * (*mpa*0 +*nqa*0+*mb_a*), max((*mb_a**(*mb_a*-1))/2, (*ppb*0 + *nqb*0)**mb_a*) + *mb_a* * *mb_a*,
nb_b * (*ppb*0 + *nqb*0 + *nb_b*)), *where*

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iarow = indxg2p (*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = *indxg2p* (*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

mpa0 = *numroc* (*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

nqa0 = *numroc* (*m*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)

iroffb = mod(*ib-1*, *mb_b*),

icoffb = mod(*jb-1*, *nb_b*),

ibrow = indxg2p (*ib*, *mb_b*, MYROW, *rsrc_b*, NPROW),

ibcol = indxg2p (*jb*, *nb_b*, MYCOL, *csrc_b*, NPCOL),

ppb0 = numroc (*p*+*iroffb*, *mb_b*, MYROW, *ibrow,* NPROW),

nqb0 = numroc (*n*+*icoffb*, *nb_b*, MYCOL, *ibcol,* NPCOL)

and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If `lwork` = -1, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a — On exit, if $m \le n$, the upper triangle of $A(ia:ia+m-1, ja+n-m:ja+n-1)$ contains the *m* by *m* upper triangular matrix R; if $m \ge n$, the elements on and above the (*m*-*n*)-th subdiagonal contain the *m* by *n* upper trapezoidal matrix *R*; the remaining elements, with the array *taua*, represent the orthogonal/unitary matrix *Q* as a product of min(*n*,*m*) elementary reflectors. (See *Application Notes* below).

taua, taub — (local)
`REAL` for `psggqrf`
`DOUBLE PRECISION` for `pdggqrf`
`COMPLEX` for `pcggqrf`
`DOUBLE COMPLEX` for `pzggqrf`.
Arrays, `DIMENSION` $LOCr(ia+m-1)$for *taua* and $LOCc(jb+\min(p,n)-1)$ for *taub*.
The array *taua* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q*.*taua* is tied to the distributed matrix *A*.(See *Application Notes* below).

The array *taub* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Z*.*taub* is tied to the distributed matrix *B*.(See *Application Notes* below).

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) `INTEGER`.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

$Q = H(ia)\, H(ia+1) \ldots H(ia+k-1)$,

where $k = \min(m,n)$.

Each $H(i)$ has the form

$H(i) = i - taua * v * v'$

where *taua* is a real/complex scalar, and *v* is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$; $v(1:n-k+i-1)$ is stored on exit in $A(ia+m-k+i-1, ja:ja+n-k+i-2)$, and *taua* in $taua(ia+m-k+i-1)$.To form *Q* explicitly, use ScaLAPACK subroutine p?orgrq/p?ungrq. To use *Q* to update another matrix, use ScaLAPACK subroutine p?ormrq/p?unmrq.

The matrix *Z* is represented as a product of elementary reflectors

$Z = H(jb)\ H(jb+1) \ldots H(jb+k-1)$,
where $k = \min(p,n)$.

Each $H(i)$ has the form

$H(i) = i - taub * v * v'$

where *taub* is a real/complex scalar, and *v* is a real/complex vector with $v(1:i-1) = 0$ and $v(i)= 1$; $v(i+1:p)$ is stored on exit in $B(ib+i:ib+p-1,jb+i-1)$, and *taub* in $taub(jb+i-1)$.To form *Z* explicitly, use ScaLAPACK subroutine p?orgqr/p?ungqr.To use *Z* to update another matrix, use ScaLAPACK subroutine p?ormqr/p?unmqr.

Symmetric Eigenproblems

To solve a symmetric eigenproblem with ScaLAPACK, you usually need to reduce the matrix to real tridiagonal form *T* and then find the eigenvalues and eigenvectors of the tridiagonal matrix *T*. ScaLAPACK includes routines for reducing the matrix to a tridiagonal form by an orthogonal (or unitary) similarity transformation $A = QTQ^H$ as well as for solving tridiagonal symmetric eigenvalue problems. These routines are listed in Table 6-4.

There are different routines for symmetric eigenproblems, depending on whether you need eigenvalues only or eigenvectors as well, and on the algorithm used (either the *QR* algorithm, or bisection followed by inverse iteration).

Table 6-4 Computational Routines for Solving Symmetric Eigenproblems

Operation	Dense symmetric/Hermitian matrix	Orthogonal/ unitary matrix	Symmetric tridiagonal matrix
Reduce to tridiagonal form $A = QTQ^H$	`p?sytrd`/`p?hetrd`		
Multiply matrix after reduction		`p?ormtr`/`p?unmtr`	
Find all eigenvalues and eigenvectors of a tridiagonal matrix *T* by a *QR* method			`p?steqr2`*)
Find selected eigenvalues of a tridiagonal matrix *T* via bisection			`p?stebz`
Find selected eigenvectors of a tridiagonal matrix *T* by inverse iteration			`p?stein`

*) This routine will be described as part of auxiliary ScaLAPACK routines.

p?sytrd

Reduces a symmetric matrix to real symmetric tridiagonal form by an orthogonal similarity transformation.

Syntax

```
call pssytrd ( uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info )
```

```
call pdsytrd ( uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info )
```

Description

This routine reduces a real symmetric matrix sub(*A*) to symmetric tridiagonal form *T* by an orthogonal similarity transformation:

$$Q' \, \mathrm{sub}(A) * Q = T,$$

where sub(*A*) = *A*(*ia*:*ia*+*n*-1,*ja*:*ja*+*n*-1).

Input Parameters

uplo (global) CHARACTER.

Specifies whether the upper or lower triangular part of the symmetric matrix sub(*A*) is stored:

If *uplo* = 'U', upper triangular
If *uplo* = 'L', lower triangular

n (global) INTEGER. The order of the distributed matrix sub(*A*) ($n \ge 0$).

a (local)

REAL for pssytrd
DOUBLE PRECISION for pdsytrd.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)).On entry, this array contains the local pieces of the symmetric distributed matrix sub(*A*).
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.(See *Application Notes* below).

ia, ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local)

REAL for pssytrd
DOUBLE PRECISION for pdsytrd.
Workspace array of dimension *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:

$lwork \geq \max(NB * (np + 1), 3 * NB)$

where NB = *mb_a* = *nb_a*,

np = numroc(*n*, NB, MYROW, *iarow*, NPROW),

iarow = indxg2p(*ia*, NB, MYROW, *rsrc_a*, NPROW).

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, if *uplo* = 'U', the diagonal and first superdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first superdiagonal, with the array *tau*, represent the orthogonal matrix *Q* as a product of elementary reflectors; if *uplo* = 'L', the diagonal and first subdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal matrix *Q* as a product of elementary reflectors. (See *Application Notes* below).

d (local)
REAL for pssytrd
DOUBLE PRECISION for pdsytrd.
Arrays, DIMENSION *LOCc*(*ja*+*n*-1).The diagonal elements of the tridiagonal matrix *T*:

$d(i) = A(i,i)$.

d is tied to the distributed matrix *A*.

e
(local)
REAL for pssytrd
DOUBLE PRECISION for pdsytrd.
Arrays, DIMENSION *LOCc*(*ja*+*n*-1) if *uplo* = 'U', *LOCc*(*ja*+*n*-2) otherwise.
The off-diagonal elements of the tridiagonal matrix *T*:

e(*i*) = *A*(*i*, *i*+1) if *uplo* = 'U',
e(*i*) = *A*(*i*+1,*i*) if *uplo* = 'L'.
e is tied to the distributed matrix *A*.

tau
(local)
REAL for pssytrd
DOUBLE PRECISION for pdsytrd.
Arrays, DIMENSION *LOCc*(*ja*+*n*-1).This array contains the scalar factors *tau* of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

work(1)
On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info
(global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

If *uplo* = 'U', 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 and *v*(*i*) = 1; *v*(1:*i*-1) is stored on exit in *A*(*ia*:*ia*+*i*-2,*ja*+*i*), and *tau* in *tau* (*ja*+*i*-1).

If *uplo* = 'L', 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 and *v*(`i`+1) = 1; *v*(`i`+2:`n`) is stored on exit in *A*(`ia`+`i`+1:`ia`+`n`-1,`ja`+`i`-1), and `tau` in `tau`(`ja`+`i`-1).

The contents of sub(*A*) on exit are illustrated by the following examples with *n* = 5:

if *uplo* = 'U':

$$\begin{bmatrix} d & e & v2 & v3 & v4 \\ & d & e & v3 & v4 \\ & & d & e & v4 \\ & & & d & e \\ & & & & d \end{bmatrix}$$

if *uplo* = 'L':

$$\begin{bmatrix} d & & & & \\ e & d & & & \\ v1 & e & d & & \\ v1 & v2 & e & d & \\ v1 & v2 & v3 & e & d \end{bmatrix}$$

where `d` and `e` denote diagonal and off-diagonal elements of *T*, and `vi` denotes an element of the vector defining *H*(`i`).

p?ormtr

Multiplies a general matrix by the orthogonal transformation matrix from a reduction to tridiagonal form determined by `p?sytrd`.

Syntax

```
call psormtr ( side, uplo, trans, m, n, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
call pdormtr ( side, uplo, trans, m, n, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
```

Description

The routine overwrites the general real distributed *m*-by-*n* matrix sub(C) = C(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side = 'L'	*side* = 'R'
trans = 'N':	Q sub(C)	sub(C) Q
trans = 'T':	Q^T sub(C)	sub(C) Q^T

where Q is a real orthogonal distributed matrix of order *nq*, with *nq* = *m* if *side* = 'L' and *nq* = *n* if *side* = 'R'. Q is defined as the product of *nq* elementary reflectors, as returned by `p?sytrd`

if *uplo* = 'U', $Q = H(nq-1)... H(2) H(1)$;

if *uplo* = 'L', $Q = H(1) H(2)... H(nq-1)$.

Input Parameters

side (global) `CHARACTER`
= `'L'`: Q or Q^T is applied from the left.
= `'R'`: Q or Q^T is applied from the right.

trans (global) `CHARACTER`
= `'N'`, no transpose, Q is applied.
= `'T'`, transpose, Q^T is applied.

uplo (global) `CHARACTER`.
= 'U': Upper triangle of A(*ia*:*,*ja*:*) contains elementary reflectors from `p?sytrd`;

= 'L': Lower triangle of A(*ia*:*,*ja*:*) contains elementary reflectors from p?sytrd

m — (global) INTEGER. The number of rows in the distributed matrix sub(C) ($m \geq 0$).

n — (global) INTEGER. The number of columns in the distributed matrix sub(C) ($n \geq 0$).

a — (local)
REAL for psormtr
DOUBLE PRECISION for pdormtr.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*m*-1)) if *side*='L', or (*lld_a*, *LOCc*(*ja*+*n*-1)) if *side* = 'R'.
Contains the vectors which define the elementary reflectors, as returned by p?sytrd.
If *side*='L', *lld_a* ≥ max(1,*LOCr*(*ia*+*m*-1));
if *side* ='R', *lld_a* ≥ max(1,*LOCr*(*ia*+*n*-1)).

ia,ja — (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix A, respectively.

desca — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix A.

tau — (local)
REAL for psormtr
DOUBLE PRECISION for pdormtr.
Array, DIMENSION of *ltau* where

if *side* = 'L' and *uplo* = 'U', *ltau* = *LOCc*(*m_a*),
if *side* = 'L' and *uplo* = 'L', *ltau* = *LOCc*(*ja*+*m*-2),
if *side* = 'R' and *uplo* = 'U', *ltau* = *LOCc*(*n_a*),
if *side* = 'R' and *uplo* = 'L', *ltau* = *LOCc*(*ja*+*n*-2). *tau*(*i*) must contain the scalar factor of the elementary reflector H(*i*), as returned by p?sytrd. *tau* is tied to the distributed matrix A.

c — (local)
REAL for psormtr
DOUBLE PRECISION for pdormtr.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)).Contains the local pieces of the distributed matrix sub (C).

work (local)
REAL for psormtr
DOUBLE PRECISION for pdormtr.
Workspace array of dimension *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:

If *uplo* = 'U',

iaa=*ia*; *jaa*=*ja*+1, *icc*=*ic*; *jcc*=*jc*;

else *uplo* = 'L',

iaa=*ia*+1, *jaa*=*ja*;

if *side* = 'L',

icc=*ic*+1; *jcc*=*jc*;

else *icc*=*ic*; *jcc*=*jc*+1;

end if

end if

If *side* = 'L',

mi=*m*-1; *ni*=*n*

lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + *mpc*0)**nb_a*) + *nb_a* * *nb_a*

else if *side* = 'R',

mi=*m*; *mi* = *n*-1;

lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max(*npa*0 + numroc(numroc(*ni*+*icoffc*, *nb_a*, 0, 0, NPCOL), * *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a* * *nb_a*

end if

where *lcmq* = *lcm* / NPCOL with *lcm* = ilcm(NPROW, NPCOL),

iroffa = mod(*iaa*-1, *mb_a*),

icoffa = mod(*jaa*-1, *nb_a*),

iarow = indxg2p (*iaa*, *mb_a*, MYROW, *rsrc_a*, NPROW),

*npa*0 = numroc(*ni*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

iroffc = mod(*icc*-1, *mb_c*),

icoffc = mod(*jcc*-1, *nb_c*),

icrow = indxg2p (*icc*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p (*jcc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*mi*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*ni*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo. If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — Overwritten by the product *Q* sub*(C)*, or *Q*'sub(*C*) or sub(*C*) *Q'* or sub(*C*) *Q*.

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?hetrd

Reduces a Hermitian matrix to Hermitian tridiagonal form by a unitary similarity transformation.

Syntax

```
call pchetrd ( uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info )
call pzhetrd ( uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info )
```

Description

This routine reduces a complex Hermitian matrix sub(*A*) to Hermitian tridiagonal form *T* by a unitary similarity transformation:

$$Q' \text{ sub}(A)\ Q = T$$

where sub(*A*) = *A*(*ia*:*ia*+*n*-1,*ja*:*ja*+*n*-1).

Input Parameters

uplo (global) CHARACTER.
Specifies whether the upper or lower triangular part of the Hermitian matrix sub(*A*) is stored:
If *uplo* = 'U', upper triangular
If *uplo* = 'L', lower triangular

n (global) INTEGER. The order of the distributed matrix sub(*A*) ($n \geq 0$).

a (local)

COMPLEX for pchetrd
DOUBLE COMPLEX for pzhetrd.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)).On entry, this array contains the local pieces of the Hermitian distributed matrix sub(*A*).
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.(See *Application Notes* below).

ia, ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local)

COMPLEX for pchetrd
DOUBLE COMPLEX for pzhetrd.
Workspace array of dimension *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:

lwork ≥ max(NB * (*np* +1), 3 * NB)

where NB = *mb_a* = *nb_a*,

np = numroc(*n*, NB, MYROW, *iarow*, NPROW),

iarow = indxg2p(*ia*, NB, MYROW, *rsrc_a*, NPROW).

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — On exit, if *uplo* = 'U', the diagonal and first superdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first superdiagonal, with the array *tau*, represent the unitary matrix *Q* as a product of elementary reflectors; if *uplo* = 'L', the diagonal and first subdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements below the first subdiagonal, with the array *tau*, represent the unitary matrix *Q* as a product of elementary reflectors. (See *Application Notes* below).

d — (local)
REAL for pchetrd
DOUBLE PRECISION for pzhetrd.
Arrays, DIMENSION *LOCc*(*ja*+*n*-1).The diagonal elements of the tridiagonal matrix *T*:

d(*i*) = *A*(*i*, *i*).

d is tied to the distributed matrix *A*.

e — (local)
REAL for pchetrd
DOUBLE PRECISION for pzhetrd.
Arrays, DIMENSION *LOCc*(*ja*+*n*-1) if *uplo* = 'U', *LOCc*(*ja*+*n*-2) otherwise.
The off-diagonal elements of the tridiagonal matrix *T*:

	e(*i*) = *A*(*i*, *i*+1) if *uplo* = 'U', *e*(*i*) = *A*(*i*+1,*i*) if *uplo* = 'L'. *e* is tied to the distributed matrix *A*.
tau	(local) COMPLEX for pchetrd DOUBLE COMPLEX for pzhetrd. Arrays, DIMENSION *LOCc*(*ja*+*n*-1).This array contains the scalar factors *tau* of the elementary reflectors. *tau* is tied to the distributed matrix *A*.
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

If *uplo* = 'U', the matrix *Q* is represented as a product of elementary reflectors

$Q = H(n-1) \dots H(2) H(1)$.

Each $H(i)$ has the form

$H(i) = i - tau * v * v'$,

where *tau* is a complex scalar, and *v* is a complex vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-2, ja+i)$, and *tau* in *tau* $(ja+i-1)$.

If *uplo* = 'L', the matrix *Q* is represented as a product of elementary reflectors

$Q = H(1) H(2) \dots H(n-1)$.

Each $H(i)$ has the form

$H(i) = i - tau * v * v'$,

where *tau* is a complex scalar, and *v* is a complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and *tau* in $tau(ja+i-1)$.

The contents of sub(A) on exit are illustrated by the following examples with $n = 5$:

if *uplo* = 'U':

$$\begin{bmatrix} d & e & v2 & v3 & v4 \\ & d & e & v3 & v4 \\ & & d & e & v4 \\ & & & d & e \\ & & & & d \end{bmatrix}$$

if *uplo* = 'L':

$$\begin{bmatrix} d & & & & \\ e & d & & & \\ v1 & e & d & & \\ v1 & v2 & e & d & \\ v1 & v2 & v3 & e & d \end{bmatrix}$$

where *d* and *e* denote diagonal and off-diagonal elements of *T*, and *vi* denotes an element of the vector defining *H*(*i*).

p?unmtr

Multiplies a general matrix by the unitary transformation matrix from a reduction to tridiagonal form determined by `p?hetrd`.

Syntax

```
call pcunmtr ( side, uplo, trans, m, n, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
call pzunmtr ( side, uplo, trans, m, n, a, ia, ja, desca, tau, c, ic, jc,
    descc, work, lwork, info )
```

Description

The routine overwrites the general complex distributed *m*-by-*n* matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side = 'L'	*side* = 'R'
trans = 'N':	*Q* sub(*C*)	sub(*C*) Q
trans = 'C':	Q^H sub(C)	sub(*C*) Q^H

where *Q* is a complex unitary distributed matrix of order *nq*, with *nq* = *m* if *side* = 'L' and *nq* = *n* if *side* = 'R'. *Q* is defined as the product of *nq*-1 elementary reflectors, as returned by p?hetrd

if *uplo* = 'U', *Q* = *H*(*nq*-1)... *H*(2) *H*(1);

if *uplo* = 'L', *Q* = *H*(1) *H*(2)... *H*(*nq*-1).

Input Parameters

side (global) CHARACTER
='L': *Q* or Q^H is applied from the left.
='R': *Q* or Q^H is applied from the right.

trans (global) CHARACTER
='N', no transpose, *Q* is applied.
='C', conjugate transpose, Q^H is applied.

uplo (global) CHARACTER.
= 'U': Upper triangle of *A*(*ia*:*,*ja*:*) contains elementary reflectors from p?hetrd;
= 'L': Lower triangle of *A*(*ia*:*,*ja*:*) contains elementary reflectors from p?hetrd

m (global) INTEGER. The number of rows in the distributed matrix sub(*C*) ($m \geq 0$).

n (global) INTEGER. The number of columns in the distributed matrix sub(*C*) ($n \geq 0$).

a (local)
REAL for pcunmtr
DOUBLE PRECISION for pzunmtr.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*m*-1)) if *side*='L', or (*lld_a*, *LOCc*(*ja*+*n*-1)) if *side* = 'R'.
Contains the vectors which define the elementary reflectors, as returned by

p?hetrd.
If *side*='L', *lld_a* ≥ max(1,*LOCr*(*ia*+*m*-1));
if *side* ='R', *lld_a* ≥ max(1,*LOCr*(*ia*+*n*-1)).

ia,ja — (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau — (local)
COMPLEX for pcunmtr
DOUBLE COMPLEX for pzunmtr.
Array, DIMENSION of *ltau* where

if *side* = 'L' and *uplo* = 'U', *ltau* = *LOCc*(*m_a*),
if *side* = 'L' and *uplo* = 'L', *ltau* = *LOCc*(*ja*+*m*-2),
if *side* = 'R' and *uplo* = 'U', *ltau* = *LOCc*(*n_a*),
if *side* = 'R' and *uplo* = 'L', *ltau* = *LOCc*(*ja*+*n*-2). *tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by p?hetrd. *tau* is tied to the distributed matrix *A*.

c — (local)
COMPLEX for pcunmtr
DOUBLE COMPLEX for pzunmtr.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)).Contains the local pieces of the distributed matrix sub (*C*).

work — (local)
COMPLEX for pcunmtr
DOUBLE COMPLEX for pzunmtr.
Workspace array of dimension *lwork*.

lwork — (local or global) INTEGER, dimension of *work*, must be at least:

If *uplo* = 'U',

iaa=*ia*; *jaa*=*ja*+1, *icc*=*ic*; *jcc*=*jc*;

else *uplo* = 'L',

iaa=*ia*+1, *jaa*=*ja*;

if *side* = 'L',

icc=*ic*+1; *jcc*=*jc*;

else *icc*=*ic*; *jcc*=*jc*+1;

end if

end if

If *side* = 'L',

mi=*m*-1; *ni*=*n*

lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + *mpc*0)**nb_a*) + *nb_a* * *nb_a*

else if *side* = 'R',

mi=*m*; *mi* = *n*-1;

lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max(*npa*0 + numroc(numroc(*ni*+*icoffc*, *nb_a*, 0, 0, NPCOL), * *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a* * *nb_a*

end if

where *lcmq* = *lcm* / NPCOL with *lcm* = ilcm(NPROW, NPCOL),

iroffa = mod(*iaa*-1, *mb_a*),

icoffa = mod(*jaa*-1, *nb_a*),

iarow = indxg2p (*iaa*, *mb_a*, MYROW, *rsrc_a*, NPROW),

*npa*0 = numroc(*ni*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

iroffc = mod(*icc*-1, *mb_c*),

icoffc = mod(*jcc*-1, *nb_c*),

icrow = indxg2p (*icc*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p (*jcc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*mi*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*ni*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo. If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — Overwritten by the product *Q* sub(*C*), or *Q*'sub(*C*) or sub(*C*) *Q'* or sub(*C*) *Q*.

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -*i*

p?stebz

Computes the eigenvalues of a symmetric tridiagonal matrix by bisection.

Syntax

```
call psstebz ( ictxt, range, order, n, vl, vu, il, iu, abstol, d, e, m,
    nsplit, w, iblock, isplit, work, iwork, liwork, info)
call pdstebz ( ictxt, range, order, n, vl, vu, il, iu, abstol, d, e, m,
    nsplit, w, iblock, isplit, work, iwork, liwork, info)
```

Description

This routine computes the eigenvalues of a symmetric tridiagonal matrix in parallel. These may be all eigenvalues, all eigenvalues in the interval

$[vl\ vu]$, or the eigenvalues indexed *il* through *iu*. A static partitioning of work is done at the beginning of p?stebz which results in all processes finding an (almost) equal number of eigenvalues.

Input Parameters

ictxt — (global) INTEGER.
The BLACS context handle.

range — (global) CHARACTER. Must be 'A' or 'V' or 'I'.
If *range* = 'A', the routine computes all eigenvalues.
If *range* = 'V', the routine computes eigenvalues in the interval $[vl\ vu]$

If *range* ='I', the routine computes eigenvalues with indices *il* to *iu*.

order
(global) CHARACTER. Must be 'B' or 'E'.
If *order* ='B', the eigenvalues are to be ordered from smallest to largest within each split-off block.
If *order* ='E', the eigenvalues for the entire matrix are to be ordered from smallest to largest.

n
(global) INTEGER. The order of the tridiagonal matrix T ($n \ge 0$).

vl, vu
(global)
REAL for psstebz
DOUBLE PRECISION for pdstebz.
If *range* ='V', the routine computes the lower and the upper bounds for the eigenvalues on the interval $[vl\ vu]$.

If *range* ='A' or 'I', *vl* and *vu* are not referenced.

il, iu
(global)
INTEGER. Constraint: $1 \le il \le iu \le n$.
If *range* ='I', the index of the smallest eigenvalue is returned for *il* and of the largest eigenvalue for *iu* (assuming that the eigenvalues are in ascending order) must be returned.
il must be at least 1. *iu* must be at least *il* and no greater than *n*.

If *range* ='A' or 'V', *il* and *iu* are not referenced.

abstol
(global)
REAL for psstebz
DOUBLE PRECISION for pdstebz.
The absolute tolerance to which each eigenvalue is required. An eigenvalue (or cluster) is considered to have converged if it lies in an interval of width *abstol*. If $abstol \le 0$, then the tolerance is taken as $ulp||T||$, where *ulp* is the machine precision and $||T||$ means the 1-norm of T

Eigenvalues will be computed most accurately when *abstol* is set to the underflow threshold slamch('U'), not 0.
Note that if eigenvectors are desired later by inverse iteration (p?stein), *abstol* should be set to 2*p?lamch('S').

d
(global)
REAL for psstebz
DOUBLE PRECISION for pdstebz.

Array, DIMENSION (*n*).

Contains *n* diagonal elements of the tridiagonal matrix *T*. To avoid overflow, the matrix must be scaled so that its largest entry is no greater than the overflow$^{(1/2)}$ * underflow$^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.

e (global)
`REAL` for `psstebz`
`DOUBLE PRECISION` for `pdstebz`.
Array, `DIMENSION` (*n* - 1).

Contains (*n*-1) off-diagonal elements of the tridiagonal matrix *T*. To avoid overflow, the matrix must be scaled so that its largest entry is no greater than overflow$^{(1/2)}$ * underflow$^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.

work (local)
`REAL` for `psstebz`
`DOUBLE PRECISION` for `pdstebz`.
Array, `DIMENSION` max(5*n*, 7). This is a workspace array.

lwork (local) `INTEGER`.
the size of the *work* array must be ≥ `max`(5*n*, 7).

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

iwork (local) `INTEGER`.
Array, `DIMENSION` max(4*n*, 14). This is a workspace array.

liwork (local) `INTEGER`.
the size of the *iwork* array must be ≥ `max`(4*n*, 14, `NPROCS`).

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

m (global) `INTEGER`. The actual number of eigenvalues found. $0 \le m \le n$

nsplit (global) `INTEGER`. The number of diagonal blocks detected in *T*.
$1 \le nsplit \le n$

w (global) REAL for psstebz
DOUBLE PRECISION for pdstebz.
Array, DIMENSION (*n*).
On exit, the first *m* elements of *w* contain the eigenvalues on all processes.

iblock (global)
INTEGER.
Array, DIMENSION (*n*).
At each row/column *j* where *e*(*j*) is zero or small, the matrix *T* is considered to split into a block diagonal matrix. On exit *iblock*(*i*) specifies which block (from 1 to the number of blocks) the eigenvalue *w*(*i*) belongs to.

NOTE. In the (theoretically impossible) event that bisection does not converge for some or all eigenvalues, *info* is set to 1 and the ones for which it did not are identified by a negative block number.

isplit (global)
INTEGER.
Array, DIMENSION (*n*).
Contains the splitting points, at which *T* breaks up into submatrices. The first submatrix consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), etc., and the *nsplit*-th consists of rows/columns *isplit*(*nsplit*-1)+1 through *isplit*(*nsplit*)=*n*. (Only the first *nsplit* elements are used, but since the *nsplit* values are not known, *n* words must be reserved for *isplit*.)

info (global)
INTEGER.
If *info* = 0, the execution is successful.
If *info* < 0, if *info* = -*i*, the *i*-th argument has an illegal value.
If *info* > 0, some or all of the eigenvalues fail to converge or not computed.
If *info* = 1, bisection fails to converge for some eigenvalues; these eigenvalues are flagged by a negative block number. The effect is that the eigenvalues may not be as accurate as the absolute and relative tolerances.
If *info* = 2, mismatch between the number of eigenvalues output and the number desired.
If *info* = 3: *range*='*i*', and the Gershgorin interval initially used is incorrect. No eigenvalues are computed. Probable cause: the machine has a sloppy floating point arithmetic. Increase the *fudge* parameter, recompile, and try again.

p?stein

Computes the eigenvectors of a tridiagonal matrix using inverse iteration.

Syntax

```
call psstein ( n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz,
    work, lwork, iwork, liwork, ifail, iclustr, gap, info)
call pdstein ( n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz,
    work, lwork, iwork, liwork, ifail, iclustr, gap, info)
call pcstein ( n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz,
    work, lwork, iwork, liwork, ifail, iclustr, gap, info)
call pzstein ( n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz,
    work, lwork, iwork, liwork, ifail, iclustr, gap, info)
```

Description

This routine computes the eigenvectors of a symmetric tridiagonal matrix T corresponding to specified eigenvalues, by inverse iteration. `p?stein` does not orthogonalize vectors that are on different processes. The extent of orthogonalization is controlled by the input parameter *`lwork`*. Eigenvectors that are to be orthogonalized are computed by the same process. `p?stein` decides on the allocation of work among the processes and then calls `sstein2` (modified LAPACK routine) on each individual process. If insufficient workspace is allocated, the expected orthogonalization may not be done.

NOTE. If the eigenvectors obtained are not orthogonal, increase *`lwork`* and run the code again.

p = `NPROW` * `NPCOL` is the total number of processes.

Input Parameters

n (global) `INTEGER`. The order of the matrix T ($n \geq 0$).

m (global) `INTEGER`. The number of eigenvectors to be returned.

d, e, w (global)
REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Arrays:
d(*) contains the diagonal elements of *T*.
DIMENSION (*n*).

e(*) contains the off-diagonal elements of *T*.
DIMENSION (*n*-1).

w(*) contains all the eigenvalues grouped by split-off block.The eigenvalues are supplied from smallest to largest within the block. (Here the output array *w* from p?stebz with order = 'B' is expected. The array should be replicated in all processes.
DIMENSION(*m*)

iblock (global) INTEGER.
Array, DIMENSION (*n*).
The submatrix indices associated with the corresponding eigenvalues in *w* -- 1 for eigenvalues belonging to the first submatrix from the top, 2 for those belonging to the second submatrix, etc. (The output array *iblock* from p?stebz is expected here).

isplit (global) INTEGER.
Array, DIMENSION (*n*).
The splitting points, at which *T* breaks up into submatrices. The first submatrix consists of rows/columns 1 to *isplit* (1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), etc., and the *nsplit*-th consists of rows/columns *isplit* (*nsplit*-1)+1 through *isplit*(*nsplit*)=*n* (The output array *isplit* from p?stebz is expected here.)

orfac (global)
REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors. *orfac* specifies which eigenvectors should be orthogonalized. Eigenvectors that correspond to eigenvalues within *orfac**‖T‖ of each other are to be orthogonalized. However, if the workspace is insufficient (see *lwork*), this tolerance may be decreased until all eigenvectors can be stored in one process. No orthogonalization is done if *orfac* is equal to zero. A default value of 10^3 is used if *orfac* is negative. *orfac* should be identical on all processes

iz, jz (global) INTEGER. The row and column indices in the global array *z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *Z*.

work (local).
REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors. Workspace array, DIMENSION (*lwork*).

lwork (local) INTEGER.
lwork controls the extent of orthogonalization which can be done. The number of eigenvectors for which storage is allocated on each process is

nvec = floor((*lwork*- max(5**n*,*np*00**mq*00))/*n*). Eigenvectors corresponding to eigenvalue clusters of size *nvec*- ceil(*m*/*p*) + 1 are guaranteed to be orthogonal (the orthogonality is similar to that obtained from?stein2).

NOTE. *lwork* must be no smaller than:
max(5**n*,*np*00**mq*00) + ceil(*m*/*p*)**n*,
and should have the same input value on all processes.

It is the minimum value of *lwork* input on different processes * that is significant.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

iwork (local) INTEGER.
Workspace array, DIMENSION (3*n*+*p*+1).

liwork (local) INTEGER. The size of the array *iwork*. It must be ≥ 3**n* + *p* + 1.

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

z (local)
REAL for psstein
DOUBLE PRECISION for pdstein
COMPLEX for pcstein
DOUBLE COMPLEX for pzstein.
Array, DIMENSION (*descz*(*dlen_*), *n*/NPCOL + NB).*z* contains the computed eigenvectors associated with the specified eigenvalues. Any vector which fails to converge is set to its current iterate after MAXIT iterations (See?stein2). On output, *z* is distributed across the *p* processes in block cyclic format.

work(1) On exit, *work*(1) gives a lower bound on the workspace (*lwork*) that guarantees the user desired orthogonalization (see *orfac*). Note that this may overestimate the minimum workspace needed.

iwork On exit, *iwork*(1) contains the amount of integer workspace required. On exit, the *iwork*(2) through *iwork*(*p*+2) indicate the eigenvectors computed by each process. Process *i* computes eigenvectors indexed *iwork*(*i*+2)+1 through *iwork*(*i*+3).

ifail (global).
INTEGER. Array, DIMENSION (*m*).
On normal exit, all elements of *ifail* are zero. If one or more eigenvectors fail to converge after MAXIT iterations (as in ?stein), then *info* > 0 is returned. If mod(*info*,*m*+1)>0, then for *i*=1 to mod(*info*,*m*+1), the eigenvector corresponding to the eigenvalue *w* (*ifail*(*i*)) failed to converge (*w* refers to the array of eigenvalues on output).

iclustr (global) INTEGER. Array, DIMENSION (2**p*)
This output array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be orthogonalized due to insufficient workspace (see *lwork, orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2*I-1) to *iclustr*(2*I), *i* = 1 to *info*/(*m*+1), could not be orthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these * clusters may not be orthogonal. *iclustr* is a zero terminated array --- (*iclustr*(2**k*).*ne.0.and*. *iclustr*(2**k*+1).*eq.0*) if and only if *k* is the number of clusters.

gap (global)

REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
This output array contains the gap between eigenvalues whose eigenvectors

could not be orthogonalized. The *info*/*m* output values in this array correspond to the *info*/(*m*+1) clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the i^{th} cluster may be as high as (*O*(*n*)**macheps*) / *gap*(*i*).

info (global) INTEGER.
If *info* = 0, the execution is successful.
If *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
If *info* < 0: if *info* = -*i*, the *i*-th argument had an illegal value.
If *info*> 0: if mod(*info*,*m*+1) = *i*, then *i* eigenvectors failed to converge in MAXIT iterations. Their indices are stored in the array *ifail*. if *info*/(*m*+1) = *i*, then eigenvectors corresponding to *i* clusters of eigenvalues could not be orthogonalized due to insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

Nonsymmetric Eigenvalue Problems

This section describes ScaLAPACK routines for solving nonsymmetric eigenvalue problems, computing the Schur factorization of general matrices, as well as performing a number of related computational tasks.
For the definition of the nonsymmetric eigenproblem, see Chapter 5.

To solve a nonsymmetric eigenvalue problem with ScaLAPACK, you usually need to reduce the matrix to the upper Hessenberg form and then solve the eigenvalue problem with the Hessenberg matrix obtained.

Table 6-5 lists ScaLAPACK routines for reducing the matrix to the upper Hessenberg form by an orthogonal (or unitary) similarity transformation
$A = QHQ^H$, as well as routines for solving eigenproblems with Hessenberg matrices, and multiplying the matrix after reduction.

Table 6-5 Computational Routines for Solving Nonsymmetric Eigenproblems

Operation performed	General matrix	Orthogonal/unitary matrix	Hessenberg matrix
Reduce to Hessenberg form $A = QHQ^H$	p?gehrd		
Multiply the matrix after reduction		p?ormhr/p?unmhr	
Find eigenvalues and Schur factorization			p?lahqr

p?gehrd

Reduces a general matrix to upper Hessenberg form.

Syntax

```
call psgehrd ( n, ilo, ihi, a, ia, ja, desca, tau, work, lwork,
     info )
call pdgehrd ( n, ilo, ihi, a, ia, ja, desca, tau, work, lwork,
     info )
call pcgehrd ( n, ilo, ihi, a, ia, ja, desca, tau, work, lwork,
     info )
call pzgehrd ( n, ilo, ihi, a, ia, ja, desca, tau, work, lwork,
     info )
```

Description

The routine reduces a real/complex general distributed matrix sub (A) to upper Hessenberg form H by an orthogonal or unitary similarity transformation

$$Q' \text{sub}(A)\, Q = H,$$

where sub(A) = A(*ia*+*n*-1:*ia*+*n*-1,*ja*+*n*-1:*ja*+*n*-1).

Input Parameters

n (global) `INTEGER`. The order of the distributed matrix sub(A) ($n \geq 0$).

ilo, ihi (global) `INTEGER`. It is assumed that sub(A) is already upper triangular in rows *ia*:*ia*+*ilo*-2 and *ia*+*ihi*:*ia*+*n*-1 and columns *ja*:*ja*+*ilo*-2 and *ja*+*ihi*:*ja*+*n*-1. (See *Application Notes* below).
If $n > 0$, $1 \leq ilo \leq ihi \leq n$; otherwise set *ilo* = 1, *ihi* = *n*.

a (local)
`REAL` for `psgehrd`
`DOUBLE PRECISION` for `pdgehrd`
`COMPLEX` for `pcgehrd`
`DOUBLE COMPLEX` for `pzgehrd`.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, this array contains the local pieces of the *n*-by-*n* general distributed matrix sub(A) to be reduced.

ia, ja — (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

work — (local)
REAL for psgehrd
DOUBLE PRECISION for pdgehrd
COMPLEX for pcgehrd
DOUBLE COMPLEX for pzgehrd.
Workspace array of dimension *lwork*.

lwork — (local or global) INTEGER, dimension of the array *work*. *lwork* is local input and must be at least

lwork ≥ NB*NB + NB*max(*ihip*+1, *ihlp*+*inlq*)

where NB = *mb_a* = *nb_a*,
iroffa = mod(*ia*-1, NB),
icoffa = mod(*ja*-1, NB),
ioff = mod(*ia*+*ilo*-2, NB),
iarow = indxg2p(*ia*, NB, MYROW, *rsrc_a*, NPROW), *ihip* = numroc(*ihi*+*iroffa*, NB, MYROW, *iarow*, NPROW),
ilrow = indxg2p(*ia*+*ilo*-1, NB, MYROW, *rsrc_a*, NPROW),
ihlp = numroc(*ihi*-*ilo*+*ioff*+1, NB,MYROW, *ilrow*, NPROW),
ilcol = indxg2p(*ja*+*ilo*-1, NB, MYCOL, *csrc_a*, NPCOL),
inlq = numroc(*n*-*ilo*+*ioff*+1, NB, MYCOL, *ilcol*, NPCOL),

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — On exit, the upper triangle and the first subdiagonal of sub(*A*) are overwritten with the upper Hessenberg matrix *H*, and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors. (See *Application Notes* below).

tau (local).
REAL for psgehrd
DOUBLE PRECISION for pdgehrd
COMPLEX for pcgehrd
DOUBLE COMPLEX for pzgehrd.
Array, DIMENSION at least max (*ja*+*n*-2).
The scalar factors of the elementary reflectors (*see Application Notes* below). Elements *ja*:*ja*+*ilo*-2 and *ja*+*ihi*:*ja*+*n*-2 of *tau* are set to zero. *tau* is tied to the distributed matrix *A*.

work(1) On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of (*ihi*-*ilo*) elementary reflectors

Q = *H*(*ilo*) *H*(*ilo*+1)... *H*(*ihi*-1).

Each *H*(*i*) has the form

H(*i*) = *i* - *tau* * *v* * *v'*

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1:*i*) = 0, *v*(*i*+1) = 1 and *v*(*ihi*+1:*n*) = 0; *v*(*i*+2:*ihi*) is stored on exit in *a*(*ia*+*ilo*+*i*:*ia*+*ihi*-1,*ja*+*ilo*+*i*-2), and *tau* in *tau*(*ja*+*ilo*+*i*-2). The contents of *a*(*ia*:*ia*+*n*-1,*ja*:*ja*+*n*-1) are illustrated by the following example, with *n* = 7, *ilo* = 2 and *ihi* = 6:
on entry

$$\begin{bmatrix} a & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & & & & & & a \end{bmatrix}$$

on exit

$$\begin{bmatrix} a & a & h & h & h & h & a \\ & a & h & h & h & h & a \\ & h & h & h & h & h & h \\ & v2 & h & h & h & h & h \\ & v2 & v3 & h & h & h & h \\ & v2 & v3 & v4 & h & h & h \\ & & & & & & a \end{bmatrix}$$

where *a* denotes an element of the original matrix sub(*A*), *H* denotes a modified element of the upper Hessenberg matrix *H*, and *vi* denotes an element of the vector defining *H*(*ja*+*ilo*+*i*-2).

p?ormhr

Multiplies a general matrix by the orthogonal transformation matrix from a reduction to Hessenberg form determined by `p?gehrd`.

Syntax

```
call psormhr ( side, trans, m, n, ilo, ihi, a, ia, ja, desca, tau, c, ic,
    jc, descc, work, lwork, info )
call pdormhr ( side, trans, m, n, ilo, ihi, a, ia, ja, desca, tau, c, ic,
    jc, descc, work, lwork, info )
```

Description

The routine overwrites the general real distributed *m*-by-*n* matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side = 'L'	*side* = 'R'
trans = 'N':	Q sub(C)	sub(C) Q
trans = 'T':	Q^T sub(C)	sub(C) Q^T

where Q is a real orthogonal distributed matrix of order *nq*, with *nq* = *m* if *side* = 'L' and *nq* = *n* if *side* = 'R'. Q is defined as the product of *ihi*-*ilo* elementary reflectors, as returned by p?gehrd.

$Q = H($*ilo*$)\, H($*ilo*$+1) \ldots H($*ihi*$-1)$.

Input Parameters

side (global) CHARACTER
='L': Q or Q^T is applied from the left.
='R': Q or Q^T is applied from the right.

trans (global) CHARACTER
='N', no transpose, Q is applied.
='T', transpose, Q^T is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub (C) ($m \geq 0$).

n (global) INTEGER. The number of columns in he distributed matrix sub (C) ($n \geq 0$).

ilo, ihi (global) INTEGER.
ilo and *ihi* must have the same values as in the previous call of p?gehrd. Q is equal to the unit matrix except for the distributed submatrix $Q(ia+ilo:ia+ihi-1, ia+ilo:ja+ihi-1)$.

If *side* = 'L', $1 \leq ilo \leq ihi \leq \max(1,m)$;
if *side* = 'R', $1 \leq ilo \leq ihi \leq \max(1,n)$;
ilo and *ihi* are relative indexes.

a (local)
REAL for psormhr
DOUBLE PRECISION for pdormhr
Pointer into the local memory to an array of dimension (*lld_a*, $LOCc(ja+m-1)$) if *side*='L', and (*lld_a*, $LOCc(ja+n-1)$) if *side* = 'R'.
Contains the vectors which define the elementary reflectors, as returned by p?gehrd.

ia, ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix A, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix A.

tau (local)
REAL for psormhr
DOUBLE PRECISION for pdormhr
Array, DIMENSION *LOCc*(*ja*+*m*-2),if *side* = 'L',
and *LOCc*(*ja*+*n*-2) if *side* = 'R'.
This array contains the scalar factors *tau*(*j*) of the elementary reflectors *H*(*j*) as returned by p?gehrd. *tau* is tied to the distributed matrix *A*.

c (local)
REAL for psormhr
DOUBLE PRECISION for pdormhr
Pointer into the local memory to an array of dimension (*lld_c*,*LOCc*(*jc*+*n*-1)). Contains the local pieces of the distributed matrix sub(*C*).

ic,jc (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)
REAL for psormhr
DOUBLE PRECISION for pdormhr
Workspace array with dimension *lwork*.

lwork (local or global) INTEGER.
The dimension of the array *work*.
lwork must be at least
iaa = *ia* + *ilo*; *jaa* = *ja*+*ilo*-1;
if *side* = 'L',
mi = *ihi*-*ilo*; *ni* = *n*; *icc* = *ic* + *ilo*; *jcc* = *jc*; *lwork* ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + *mpc*0)**nb_a*) + *nb_a* * *nb_a*

else if *side* = 'R',
mi = *m*; *ni* = *ihi*-*ilo*; *icc* = *ic*; *jcc* = *jc* + *ilo*; *lwork* ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max(*npa*0 + numroc(numroc(*ni*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a* * *nb_a*

end if

where *lcmq* = *lcm* / NPCOL with *lcm* = ilcm(NPROW, NPCOL),

iroffa = mod(*iaa*-1, *mb_a*),
icoffa = mod(*jaa*-1, *nb_a*),
iarow = indxg2p (*iaa*, *mb_a*, MYROW, *rsrc_a*, NPROW),
*npa*0 = numroc(*ni*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
iroffc = mod(*icc*-1, *mb_c*),
icoffc = mod(*jcc*-1, *nb_c*),
icrow = indxg2p (*icc*, *mb_c*, MYROW, *rsrc_c*, NPROW),
iccol = indxg2p (*jcc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),
*mpc*0 = numroc(*mi*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),
*nqc*0 = numroc(*ni*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — sub(C) is overwritten by Q sub(C) or Q'sub(C) or sub(C)Q' or sub(C)Q.

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?unmhr

Multiplies a general matrix by the unitary transformation matrix from a reduction to Hessenberg form determined by `p?gehrd`.

Syntax

```
call pcunmhr ( side, trans, m, n, ilo, ihi, a, ia, ja, desca, tau, c, ic,
    jc, descc, work, lwork, info )
call pzunmhr ( side, trans, m, n, ilo, ihi, a, ia, ja, desca, tau, c, ic,
    jc, descc, work, lwork, info )
```

Description

The routine overwrites the general complex distributed *m*-by-*n* matrix sub(C) = C(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side = 'L'	*side* = 'R'
trans = 'N':	Q sub(C)	sub(C) Q
trans = 'C':	Q^H sub(C)	sub(C) Q^H

where Q is a complex unitary distributed matrix of order *nq*, with *nq* = *m* if *side* = 'L' and *nq* = *n* if *side* = 'R'. Q is defined as the product of *ihi*-*ilo* elementary reflectors, as returned by `p?gehrd`

Q = H(*ilo*) H(*ilo*+1)... H(*ihi*-1).

Input Parameters

side (global) `CHARACTER`
='L': Q or Q^H is applied from the left.
='R': Q or Q^H is applied from the right.

trans (global) `CHARACTER`
='N', no transpose, Q is applied.
='C', conjugate transpose, Q^H is applied.

m (global) `INTEGER`. The number of rows in the distributed submatrix sub (C) ($m \geq 0$).

n (global) `INTEGER`. The number of columns in the distributed submatrix sub (C) ($n \geq 0$).

ilo, ihi (global) `INTEGER`. These must be the same parameters *ilo* and *ihi*, respectively, as supplied to `p?gehrd`.Q is equal to the unit matrix except in the distributed submatrix
Q (*ia*+*ilo*:*ia*+*ihi*-1,*ia*+*ilo*:*ja*+*ihi*-1).
If *side* ='L', then $1 \leq ilo \leq ihi \leq$ max (1,*m*).
If *side* ='R', then $1 \leq ilo \leq ihi \leq$ max (1,*n*)
ilo and *ihi* are relative indexes.

a (local)
`COMPLEX` for `pcunmhr`
`DOUBLE COMPLEX` for `pzunmhr`.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*m*-1)) if *side*='L', and (*lld_a*, *LOCc*(*ja*+*n*-1)) if *side* = 'R'.
Contains the vectors which define the elementary reflectors, as returned by `p?gehrd.`

ia,ja (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)
`COMPLEX` for `pcunmhr`
`DOUBLE COMPLEX` for `pzunmhr`.
Array, `DIMENSION` *LOCc*(*ja*+*m*-2),if *side* = 'L',
and *LOCc*(*ja*+*n*-2) if *side* = 'R'.
This array contains the scalar factors *tau*(*j*) of the elementary reflectors *H*(*j*) as returned by `p?gehrd`. *tau* is tied to the distributed matrix *A*.

c (local)
`COMPLEX` for `pcunmhr`
`DOUBLE COMPLEX` for `pzunmhr`.
Pointer into the local memory to an array of dimension (*lld_c*, *LOCc*(*jc*+*n*-1)). Contains the local pieces of the distributed matrix sub(*C*).

ic,jc (global) `INTEGER`. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)
COMPLEX for pcunmhr
DOUBLE COMPLEX for pzunmhr.
Workspace array with dimension *lwork*.

lwork (local or global)
The dimension of the array *work*.
lwork must be at least
iaa = *ia* + *ilo*; *jaa* = *ja*+*ilo*-1;
if *side* = 'L',
mi = *ihi*-*ilo*; *ni* = *n*; *icc* = *ic* + *ilo*; *jcc* = *jc*; *lwork* ≥
max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + *mpc*0)**nb_a*) + *nb_a* * *nb_a*

else if *side* = 'R',
mi = *m*; *ni* = *ihi*-*ilo*; *icc* = *ic*; *jcc* = *jc* + *ilo*; *lwork* ≥
max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max(*npa*0 + numroc(numroc(*ni*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a* * *nb_a*

end if

where *lcmq* = *lcm* / NPCOL with *lcm* = ilcm(NPROW, NPCOL),

iroffa = mod(*iaa*-1, *mb_a*),
icoffa = mod(*jaa*-1, *nb_a*),
iarow = indxg2p (*iaa*, *mb_a*, MYROW, *rsrc_a*, NPROW),
*npa*0 = numroc(*ni*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
iroffc = mod(*icc*-1, *mb_c*),
icoffc = mod(*jcc*-1, *nb_c*),
icrow = indxg2p (*icc*, *mb_c*, MYROW, *rsrc_c*, NPROW),
iccol = indxg2p (*jcc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),
*mpc*0 = numroc(*mi*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),
*nqc*0 = numroc(*ni*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — *C* is overwritten by *Q** sub (*C*) or *Q'**sub(*C*) or sub(*C*)**Q'* or sub(*C*)**Q*.

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?lahqr

Computes the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form.

Syntax

```
call pslahqr (wantt, wantz, n, ilo, ihi, a, desca, wr, wi, iloz, ihiz, z,
    descz, work, lwork, iwork,  ilwork, info )
call pdlahqr (wantt, wantz, n, ilo, ihi, a, desca, wr, wi, iloz, ihiz, z,
    descz, work, lwork, iwork,  ilwork, info)
```

Description

This is an auxiliary routine used to find the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form from columns *ilo* to *ihi*.

Input Parameters

wantt — (global) LOGICAL.
If *wantt* = .TRUE., the full Schur form T is required;
If *wantt* = .FALSE., only eigenvalues are required.

wantz — (global) LOGICAL.
If *wantz* = .TRUE., the matrix of Schur vectors *z* is required;
If *wantz*= .FALSE., Schur vectors are not required.

n	(global) INTEGER. The order of the Hessenberg matrix *A* (and *z* if *wantz*). ($n \geq 0$).
ilo, ihi	(global) INTEGER. It is assumed that *A* is already upper quasi-triangular in rows and columns *ihi*+1:*n*, and that *A*(*ilo*,*ilo*-1) = 0 (unless *ilo* = 1). p?lahqr works primarily with the Hessenberg submatrix in rows and columns *ilo* to *ihi*, but applies transformations to all of *h* if *wantt* is .TRUE..1 ≤ *ilo* ≤ max(1,*ihi*);*ihi* ≤ *n*.
a	(global) REAL for pslahqr DOUBLE PRECISION for pdlahqr Array, DIMENSION (*desca*(*lld_*),*) .On entry, the upper Hessenberg matrix *A*.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
iloz,ihiz	(global) INTEGER. Specify the rows of *z* to which transformations must be applied if *wantz* is .TRUE.. 1 ≤ *iloz* ≤ *ilo*; *ihi* ≤ *ihiz* ≤ *n*.
z	(global) REAL for pslahqr DOUBLE PRECISION for pdlahqr Array. If *wantz* is .TRUE., on entry *z* must contain the current matrix *Z* of transformations accumulated by pdhseqr. If *wantz* is .FALSE., *z* is not referenced.
descz	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *Z*.
work	(local) REAL for pslahqr DOUBLE PRECISION for pdlahqr Workspace array with dimension *lwork*.
lwork	(local) INTEGER. The dimension of *work*. *lwork* is assumed big enough so that *lwork* ≥ 3**n* + max(2*max(*descz*(*lld_*),*desca*(*lld_*)) + 2**LOCq*(*n*), 7*ceil(*n*/*hbl*)/*lcm*(NPROW,NPCOL))). If *lwork* = -1, then *work*(1) gets set to the above number and the code returns immediately.
iwork	(global and local) INTEGER array of size *ilwork*.

ilwork (local) `INTEGER.` This holds some of the *iblk* integer arrays.

Output Parameters

a On exit, if *wantt* is `.TRUE.`, *A* is upper quasi-triangular in rows and columns *ilo*:*ihi*, with any 2-by-2 or larger diagonal blocks not yet in standard form. If *wantt* is `.FALSE.`, the contents of *A* are unspecified on exit.

work(1) On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

wr, *wi* (global replicated output)
`REAL` for `pslahqr`
`DOUBLE PRECISION` for `pdlahqr`
Arrays, `DIMENSION` (*n*) each.
The real and imaginary parts, respectively, of the computed eigenvalues *ilo* to *ihi* are stored in the corresponding elements of *wr* and *wi*. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the *i*-th and (*i*+1)-th, with *wi*(*i*) > 0 and *wi*(*i*+1) < 0. If *wantt* is `.TRUE.`, the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *A*. *A* may be returned with larger diagonal blocks until the next release.

z On exit *z* has been updated; transformations are applied only to the submatrix *z*(*iloz*:*ihiz*, *ilo*:*ihi*).

info (global) `INTEGER`.
= 0: the execution is successful.
< 0: parameter number -*info* incorrect or inconsistent
> 0: `p?lahqr` failed to compute all the eigenvalues *ilo* to *ihi* in a total of 30*(*ihi*-*ilo*+1) iterations; if *info* = *i*, elements *i*+1:*ihi* of *wr* and *wi* contain those eigenvalues which have been successfully computed.

Singular Value Decomposition

This section describes ScaLAPACK routines for computing the singular value decomposition (SVD) of a general *m* by *n* matrix *A* (see <>).

To find the SVD of a general matrix *A*, this matrix is first reduced to a bidiagonal matrix *B* by a unitary (orthogonal) transformation, and then SVD of the bidiagonal matrix is computed. Note that the SVD of *B* is computed using the LAPACK routine `?bdsqr`.

Table 6-6 lists ScaLAPACK computational routines for performing this decomposition.

Table 6-6 Computational Routines for Singular Value Decomposition (SVD)

Operation	General matrix	Orthogonal/unitary matrix
Reduce *A* to a bidiagonal matrix	`p?gebrd`	
Multiply matrix after reduction		`p?ormbr`/`p?unmbr`

p?gebrd

Reduces a general matrix to bidiagonal form.

Syntax

```
call psgebrd ( m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork,
    info )
call pdgebrd ( m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork,
    info )
call pcgebrd ( m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork,
    info )
call pzgebrd ( m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork,
    info )
```

Description

The routine reduces a real/complex general m-by-n distributed matrix sub(A) = $A(ia:ia+m-1, ja:ja+n-1)$ to upper or lower bidiagonal form B by an orthogonal/unitary transformation:

$$Q' * \text{sub}(A) * P = B.$$

If $m >= n$, B is upper bidiagonal; if $m < n$, B is lower bidiagonal.

Input Parameters

m (global) `INTEGER`. The number of rows in the distributed matrix sub(A) ($m \geq 0$).

n (global) INTEGER. The number of columns in the distributed matrix sub(*A*) ($n \geq 0$).

a (local)
REAL for psgebrd
DOUBLE PRECISION for pdgebrd
COMPLEX for pcgebrd
DOUBLE COMPLEX for pzgebrd.

Real pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, this array contains the distributed matrix sub (*A*).

ia,ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local)
REAL for psgebrd
DOUBLE PRECISION for pdgebrd
COMPLEX for pcgebrd
DOUBLE COMPLEX for pzgebrd.Workspace array of dimension *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:

lwork ≥ *nb** (*mpa*0 + nqa0+1)+ *nqa*0

where NB = *mb_a* = *nb_a*,

iroffa = mod(*ia*-1, *nb*),

icoffa = mod(*ja*-1, NB),

iarow = indxg2p (*ia*, *nb*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p (*ja*, NB, MYCOL, *csrc_a*, NPCOL),

*mpa*0 = numroc(*m* +*iroffa*, NB, MYROW, *iarow*, NPROW),

*nqa*0 = numroc(*n* +*icoffa*, NB, MYCOL, *iacol*, NPCOL),

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

if *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, if $m \geq n$, the diagonal and the first superdiagonal of sub(A) are overwritten with the upper bidiagonal matrix B; the elements below the diagonal, with the array *tauq*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and the elements above the first superdiagonal, with the array *taup*, represent the orthogonal matrix P as a product of elementary reflectors. If $m < n$, the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix B; the elements below the first subdiagonal, with the array *tauq*, represent the orthogonal/unitary 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 *Application Notes* below)

d (local)
`REAL` for single-precision flavors
`DOUBLE PRECISION` for double-precision flavors. Array, `DIMENSION` *LOCc*(*ja*+`min`(*m*, *n*)-1) if $m \geq n$; *LOCr*(*ia*+`min`(*m*, *n*)-1) otherwise. The distributed diagonal elements of the bidiagonal matrix B: $d(i) = a(i, i)$.
d is tied to the distributed matrix A.

e (local)
`REAL` for single-precision flavors
`DOUBLE PRECISION` for double-precision flavors. Array, `DIMENSION` *LOCr*(*ia*+min(*m*, *n*)-1) if $m \geq n$; *LOCc*(*ja*+`min`(*m*, *n*)-2)otherwise. The distributed off-diagonal elements of the bidiagonal distributed matrix B:

if $m \geq n$,
$e(i) = a(i, i+1)$ for $i = 1,2,...,n-1$;
if $m < n$,
$e(i) = a(i+1,i)$ for $i = 1,2,...,m-1$.
e is tied to the distributed matrix A.

tauq, *taup* (local)
`REAL` for `psgebrd`
`DOUBLE PRECISION` for `pdgebrd`
`COMPLEX` for `pcgebrd`
`DOUBLE COMPLEX` for `pzgebrd`.
Arrays, `DIMENSION` *LOCc*(*ja*+`min`(*m*, *n*)-1) for *tauq* and *LOCr*(*ia*+`min`(*m*, *n*)-1) for *taup*.
Contain the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrices Q and P, respectively. *tauq* and *taup* are tied to the distributed matrix A. (See *Application Notes* below)

work(1) On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrices Q and P are represented as products of elementary reflectors:

If $m \geq n$,

$Q = H(1) H(2)... H(n)$ and $P = G(1) G(2)... G(n-1)$.

Each $H(i)$ and $G(i)$ has the form:

$H(i) = i - tauq * v * v'$ and $G(i) = i - taup * u * u'$

where *tauq* and *taup* are real/complex scalars, and v and u are real/complex vectors;

$v(1:i-1) = 0$, $v(i) = 1$, and $v(i+1:m)$ is stored on exit in $A(ia+i:ia+m-1,ja+i-1)$;

$u(1:i) = 0$, $u(i+1) = 1$, and $u(i+2:n)$ is stored on exit in $A(ia+i-1,ja+i+1:ja+n-1)$;

tauq is stored in *tauq*($ja+i-1$) and *taup* in *taup*($ia+i-1$).

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:

$H(i) = i - tauq * v * v'$ and $G(i) = i - taup * u * u'$

where *tauq* and *taup* are real/complex scalars, and v and u are real/complex vectors;

$v(1:i) = 0$, $v(i+1) = 1$, and $v(i+2:m)$ is stored on exit in $A(ia+i:ia+m-1,ja+i-1)$; $u(1:i-1) = 0$, $u(i) = 1$, and $u(i+1:n)$ is stored on exit in $A(ia+i-1,ja+i+1:ja+n-1)$;

tauq is stored in *tauq*($ja+i-1$) and *taup* in *taup*($ia+i-1$).

The contents of sub(A) on exit are illustrated by the following examples:

$m = 6$ and $n = 5$ ($m > n$):

$$\begin{bmatrix} d & e & u1 & u1 & u1 \\ v1 & d & e & u2 & u2 \\ v1 & v2 & d & e & u3 \\ v1 & v2 & v3 & d & e \\ v1 & v2 & v3 & v4 & d \\ v1 & v2 & v3 & v4 & v5 \end{bmatrix}$$

$m = 5$ and $n = 6$ ($m < n$):

$$\begin{bmatrix} d & u1 & u1 & u1 & u1 & u1 \\ e & d & u2 & u2 & u2 & u2 \\ v1 & e & d & u3 & u3 & u3 \\ v1 & v2 & e & d & u4 & u4 \\ v1 & v2 & v3 & e & d & u5 \end{bmatrix}$$

where d and e denote diagonal and off-diagonal elements of B, vi denotes an element of the vector defining $H(i)$, and ui an element of the vector defining $G(i)$.

p?ormbr

Multiplies a general matrix by one of the orthogonal matrices from a reduction to bidiagonal form determined by `p?gebrd`.

Syntax

```
call psormbr (vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic,
    jc, descc, work, lwork, info)
call pdormbr (vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic,
    jc, descc, work, lwork, info)
```

Description

If *vect* = '*Q*', the routine overwrites the general real distributed *m*-by-*n* matrix sub(C) = C(*c*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side = 'L'	*side* = 'R'
trans = 'N':	Q sub(C)	sub(C) Q
trans = 'T':	Q^T sub(C)	sub(C) Q^T

If *vect* = 'P', the routine overwrites sub(C) with

	side = 'L'	*side* = 'R'
trans = 'N':	P sub(C)	sub(C) P
trans = 'T':	P^Tsub(C)	sub(C) P^T

Here Q and P^T are the orthogonal distributed matrices determined by `p?gebrd` when reducing a real distributed matrix A(*ia*:*,*ja*:*) to bidiagonal form: A(*ia*:*,*ja*:*) = $Q\ B\ P^T$. Q and P^T are defined as products of elementary reflectors $H(i)$ and $G(i)$ respectively.

Let *nq* = *m* if *side* = 'L' and *nq* = *n* if *side* = 'R'. Thus *nq* is the order of the orthogonal matrix Q or P^T that is applied.

If *vect* = '*Q*', A(*ia*:*,*ja*:*) is assumed to have been an *nq*-by-*k* matrix:

if $nq \geq k$, $Q = H(1)\ H(2)\ldots\ H(k)$;

if $nq < k$, $Q = H(1)\ H(2)\ldots\ H(nq\text{-}1)$.

If *vect* = '*P*', *A*(*ia*:*,*ja*:*) is assumed to have been a *k*-by-*nq* matrix:

if $k < nq$, $P = G(1)\ G(2)\ldots\ G(k)$;

if $k \ge nq$, $P = G(1)\ G(2)\ldots\ G(nq-1)$.

Input Parameters

vect (global) CHARACTER.
if *vect* ='Q', then Q or Q^T is applied.
if *vect* ='P', then P or P^T is applied.

side (global) CHARACTER.
if *side* ='L', then Q or Q^T, P or P^T is applied from the left.
if *side* ='R', then Q or Q^T, P or P^T is applied from the right.

trans (global) CHARACTER.
if *trans* ='N', no transpose, Q or P is applied.
if *trans* ='T', then Q^T or P^T is applied.

m (global)
INTEGER. The number of rows in the distributed matrix sub (*C*).

n (global) INTEGER. The number of columns in the distributed matrix sub (*C*).

k (global) INTEGER.
If *vect* = 'Q', the number of columns in the original distributed matrix reduced by p?gebrd;
If *vect* = 'P', the number of rows in the original distributed matrix reduced by p?gebrd.

Constraints: $k \ge 0$.

a (local)
REAL for psormbr
DOUBLE PRECISION for pdormbr.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+min(*nq*,*k*)-1)) if *vect*='Q',
and (*lld_a*, *LOCc*(*ja*+*nq*-1)) if *vect* = 'P'.
nq = *m* if *side* = 'L', and *nq* = *n* otherwise.
The vectors which define the elementary reflectors *H*(*i*) and *G*(*i*), whose products determine the matrices *Q* and *P*, as returned by p?gebrd.
If *vect* = 'Q', *lld_a* ≥ max(1,*LOCr*(*ia*+*nq*-1));
if *vect* = 'P', *lld_a* ≥ max(1,*LOCr*(*ia*+min(*nq*,*k*)-1)).

ia, ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psormbr
DOUBLE PRECISION for pdormbr.
Array, DIMENSION *LOCc*(*ja*+min(*nq*,*k*)-1), if *vect* = 'Q', and *LOCr*(*ia*+min(*nq*,*k*)-1), if *vect* = 'P'.
tau(*i*) must contain the scalar factor of the elementary reflector *H*(*i*) or *G*(*i*), which determines *Q* or *P*, as returned by pdgebrd in its array argument *tauq* or *taup*. *tau* is tied to the distributed matrix *A*.

c (local)
REAL for psormbr
DOUBLE PRECISION for pdormbr.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*jc*+*n*-1)). Contains the local pieces of the distributed matrix sub (*C*).

ic, jc (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)
REAL for psormbr
DOUBLE PRECISION for pdormbr.
Workspace array of dimension *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:

if *side* = 'L'

nq = *m*;

if((*vect* = 'Q' and $nq \ge k$) or (*vect* is not equal to 'Q' and $nq > k$)), *iaa*=*ia*; *jaa*=*ja*; *mi*=*m*; *ni*=*n*; *icc*=*ic*; *jcc*=*jc*;

else

iaa=*ia*+1; *jaa*=*ja*; *mi*=*m*-1; *ni*=*n*; *icc*=*ic*+1; *jcc*=*jc*;

end if

else if *side* = 'R', *nq* = *n*;

if((*vect* = 'Q' and $nq \geq k$) or (*vect* is not equal to 'Q' and $nq > k$)),

iaa=*ia*; *jaa*=*ja*; *mi*=*m*; *ni*=*n*; *icc*=*ic*; *jcc*=*jc*;

else

iaa=*ia*; *jaa*=*ja*+1; *mi*=*m*; *ni*=*n*-1; *icc*=*ic*; *jcc*=*jc*+1;

end if

end if

If *vect* = 'Q',

If *side* = 'L', *lwork* ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + *mpc*0)**nb_a*) + *nb_a* * *nb_a*

else if *side* = 'R',

lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max(*npa*0 + numroc(numroc(*ni*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a* * *nb_a* * end if

else if *vect* is not equal to 'Q', if *side* = 'L',

lwork ≥ max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + max(*mqa*0 + numroc(numroc(*mi*+iroffc, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))**mb_a*) + *mb_a* **mb_a*

else if *side* = 'R',

lwork ≥ max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + *nqc*0)**mb_a*) + *mb_a* * *mb_a*

end if

end if

where *lcmp* = *lcm* / NPROW, *lcmq* = *lcm* / NPCOL,
with *lcm* = ilcm(NPROW, NPCOL),

iroffa = mod(*iaa*-1, *mb_a*),

icoffa = mod(*jaa*-1, *nb_a*),

iarow = indxg2p (*iaa*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p (*jaa*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mqa*0 = numroc(*mi*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),

*npa*0 = numroc(*ni*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

iroffc = mod(*icc*-1, *mb_c*),

icoffc = mod(*jcc*-1, *nb_c*),

icrow = indxg2p (*icc*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p (*jcc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*mi*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*ni*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — On exit, if *vect*='Q', sub(*C*) is overwritten by *Q**sub(*C*) or *Q*'*sub(*C*) or sub(*C*)**Q*' or sub(*C*)**Q*;
if *vect*='P', sub(*C*) is overwritten by *P**sub(*C*) or *P*'*sub(*C*) or sub(*C*)**P* or sub(*C*)**P*'.

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?unmbr

Multiplies a general matrix by one of the unitary transformation matrices from a reduction to bidiagonal form determined by `p?gebrd`.

Syntax

```
call cunmbr (vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic,
    jc, descc, work, lwork, info)
call zunmbr (vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic,
    jc, descc, work, lwork, info)
```

Description

If `vect` = '*Q*', the routine overwrites the general complex distributed `m`-by-`n` matrix sub(*C*) = *C*(`ic`:`ic`+`m`-1,`jc`:`jc`+`n`-1) with

	`side` = 'L'	`side` = 'R'
`trans` = 'N':	Q sub(C)	sub(C) Q
`trans` = 'C':	Q^H sub(C)	sub(C) Q^H

If `vect` = 'P', the routine overwrites sub(*C*) with

	`side` = 'L'	`side` = 'R'
`trans` = 'N':	P sub(C)	sub(C) P
`trans` = 'C':	P^Hsub(C)	sub(C) P^H

Here Q and P^H are the unitary distributed matrices determined by `p?gebrd` when reducing a complex distributed matrix *A*(`ia`:*,`ja`:*) to bidiagonal form: *A*(`ia`:*,`ja`:*) = $Q\ B\ P^H$. Q and $P^H$are defined as products of elementary reflectors *H*(`i`) and *G*(`i`) respectively.

Let `nq` = `m` if `side` = 'L' and `nq` = `n` if `side` = 'R'. Thus `nq` is the order of the unitary matrix Q or P^H that is applied.

If `vect` = '*Q*', *A*(`ia`:*,`ja`:*) is assumed to have been an `nq`-by-`k` matrix:

if `nq` ≥ `k`, $Q = H(1)\ H(2)...\ H(k)$;

if `nq` < `k`, $Q = H(1)\ H(2)...\ H(nq-1)$.

If *vect* = '*P*', A(*ia*:*,*ja*:*) is assumed to have been a *k*-by-*nq* matrix:

if $k < nq$, $P = G(1)\ G(2)...\ G(k)$;

if $k \geq nq$, $P = G(1)\ G(2)...\ G(nq\text{-}1)$.

Input Parameters

vect (global) CHARACTER.
if *vect* = 'Q', then Q or Q^H is applied.
if *vect* = 'P', then P or P^H is applied.

side (global) CHARACTER.
if *side* = 'L', then Q or Q^H, P or P^H is applied from the left.
if *side* = 'R', then Q or Q^H, P or P^H is applied from the right.

trans (global) CHARACTER.
if *trans* = 'N', no transpose, Q or P is applied.
if *trans* = 'C', conjugate transpose, Q^H or P^H is applied.

m (global)
INTEGER. The number of rows in the distributed matrix sub (C) $m \geq 0$.

n (global) INTEGER. The number of columns in the distributed matrix sub (C) $n \geq 0$.

k (global) INTEGER.
If *vect* = 'Q', the number of columns in the original distributed matrix reduced by p?gebrd;
If *vect* = 'P', the number of rows in the original distributed matrix reduced by p?gebrd.

Constraints: $k \geq 0$.

a (local)
COMPLEX for psormbr
DOUBLE COMPLEX for pdormbr.
Pointer into the local memory to an array of dimension (*lld_a*, $LOCc$(*ja*+min(*nq*,*k*)-1)) if *vect*='Q',
and (*lld_a*, $LOCc$(*ja*+*nq*-1)) if *vect* = 'P'.
nq = *m* if *side* = 'L', and *nq* = *n* otherwise.
The vectors which define the elementary reflectors $H(i)$ and $G(i)$, whose products determine the matrices Q and P, as returned by p?gebrd.
If *vect* = 'Q', *lld_a* ≥ max(1,$LOCr$(*ia*+*nq*-1));
if *vect* = 'P', *lld_a* ≥ max(1,$LOCr$(*ia*+min(*nq*,*k*)-1)).

ia,ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)
COMPLEX for pcunmbr
DOUBLE COMPLEX for pzunmbr.
Array, DIMENSION *LOCc*(*ja*+min(*nq*,*k*)-1), if *vect* = 'Q', and *LOCr*(*ia*+min(*nq*,*k*)-1), if *vect* = 'P'.
tau(*i*) must contain the scalar factor of the elementary reflector *H*(*i*) or *G*(*i*), which determines *Q* or *P*, as returned by p?gebrd in its array argument *tauq* or *taup*. *tau* is tied to the distributed matrix *A*.

c (local)
COMPLEX for pcunmbr
DOUBLE COMPLEX for pzunmbr.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*jc*+*n*-1)). Contains the local pieces of the distributed matrix sub (*C*).

ic,jc (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)
COMPLEX for pcunmbr
DOUBLE COMPLEX for pzunmbr.
Workspace array of dimension *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:

if *side* = 'L'

nq = *m*;

if((*vect* = 'Q' and $nq \ge k$) or (*vect* is not equal to 'Q' and $nq > k$)), *iaa*=*ia*; *jaa*=*ja*; *mi*=*m*; *ni*=*n*; *icc*=*ic*; *jcc*=*jc*;

else

iaa=*ia*+1; *jaa*=*ja*; *mi*=*m*-1; *ni*=*n*; *icc*=*ic*+1; *jcc*=*jc*;

end if

else if *side* = 'R', *nq* = *n*;

if((*vect* = 'Q' and *nq* ≥ *k*) or (*vect* is not equal to 'Q' and *nq* > *k*)),

iaa=*ia*; *jaa*=*ja*; *mi*=*m*; *ni*=*n*; *icc*=*ic*; *jcc*=*jc*;

else

iaa=*ia*; *jaa*=*ja*+1; *mi*=*m*; *ni*=*n*-1; *icc*=*ic*; *jcc*=*jc*+1;

end if

end if

If *vect* = 'Q',

If *side* = 'L', *lwork* ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + *mpc*0)**nb_a*) + *nb_a* * *nb_a*

else if *side* = 'R',

lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max(*npa*0 + numroc(numroc(*ni*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a* * *nb_a* * end if

else if *vect* is not equal to 'Q', if *side* = 'L',

lwork ≥ max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + max(*mqa*0 + numroc(numroc(*mi*+iroffc, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))**mb_a*) + *mb_a* **mb_a*

else if *side* = 'R',

lwork ≥ max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + *nqc*0)**mb_a*) + *mb_a* * *mb_a*

end if

end if

where *lcmp* = *lcm* / NPROW, *lcmq* = *lcm* / NPCOL,
with *lcm* = ilcm(NPROW, NPCOL),

iroffa = mod(*iaa*-1, *mb_a*),

icoffa = mod(*jaa*-1, *nb_a*),

iarow = indxg2p (*iaa*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p (*jaa*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mqa*0 = numroc(*mi*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),

*npa*0 = numroc(*ni*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),

iroffc = mod(*icc*-1, *mb_c*),

icoffc = mod(*jcc*-1, *nb_c*),

icrow = indxg2p (*icc*, *mb_c*, MYROW, *rsrc_c*, NPROW),

iccol = indxg2p (*jcc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*mi*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*ni*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c
: On exit, if *vect*='Q', sub(*C*) is overwritten by *Q**sub(*C*) or *Q*'*sub(*C*) or sub(*C*)**Q*' or sub(*C*)**Q*;
if *vect*='P', sub(*C*) is overwritten by *P**sub(*C*) or *P*'*sub(*C*) or sub(*C*)**P* or sub(*C*)**P*'.

work(1)
: On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info
: (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Generalized Symmetric-Definite Eigenproblems

This section describes ScaLAPACK routines that allow you to reduce the *generalized symmetric-definite eigenvalue problems* (see <>) to standard symmetric eigenvalue problem $Cy = \lambda y$, which you can solve by calling ScaLAPACK routines described earlier in this chapter (see page 6-154).

Table 6-7 lists these routines.

Table 6-7 Computational Routines for Reducing Generalized Eigenproblems to Standard Problems

Operation	Real symmetric matrices	Complex Hermitian matrices
Reduce to standard problems	p?sygst	p?hegst

p?sygst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form.

Syntax

```
call pssygst ( ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb,
    scale, info )
call pdsygst ( ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb,
    scale, info )
```

Description

This routine reduces real symmetric-definite generalized eigenproblems to the standard form.

In the following sub(A) denotes A(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) and sub(B) denotes B(*ib*:*ib*+*n*-1, *jb*:*jb*+*n*-1).

If *ibtype* = 1, the problem is

$$\mathrm{sub}(A)x = \lambda\,\mathrm{sub}(B)x,$$

and sub(A) is overwritten by inv(U^T) sub(A)inv(U) or inv(L)sub(A)inv(L^T).

If *ibtype* = 2 or 3, the problem is

$$\mathrm{sub}(A)\mathrm{sub}(B)x = \lambda x \text{ or } \mathrm{sub}(B)\mathrm{sub}(A)x = \lambda x,$$

and sub(A) is overwritten by U sub(A)U^T or L^Tsub(A)L.

sub(B) must have been previously factorized as U^TU or LL^T by `p?potrf`.

Input Parameters

ibtype (global) `INTEGER`. Must be 1 or 2 or 3.
If *itype* = 1, compute inv(U^T)sub(A)inv(U) or inv(L)sub(A)inv(L^T);
If *itype* = 2 or 3, compute Usub(A)U^T or
L^Tsub(A)L.

uplo (global)
`CHARACTER`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, the upper triangle of sub(A) is stored and sub (B) is factored as U^TU.
If *uplo* = `'L'`, the lower triangle of sub (A) is stored and sub (B) is factored as LL^T.

n (global) `INTEGER`. The order of the matrices sub (A) and sub (B) ($n \geq 0$).

a (local)
`REAL` for `pssygst`
`DOUBLE PRECISION` for `pdsygst`.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the array contains the local pieces of the *n*-by-*n* symmetric distributed matrix sub(A). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(A) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(A) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

ia,ja (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix A, respectively.

desca (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix A.

b (local)
`REAL` for `pssygst`
`DOUBLE PRECISION` for `pdsygst`.
Pointer into the local memory to an array of dimension (*lld_b*, *LOCc*(*jb*+*n*-1)). On entry, the array contains the local pieces of the triangular factor from the Cholesky factorization of sub (B) as returned by `p?potrf`.

ib,jb (global) `INTEGER`. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix B, respectively.

descb (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix B.

Output Parameters

a — On exit, if *info* = 0, the transformed matrix, stored in the same format as sub(*A*).

scale — (global)
REAL for pssygst
DOUBLE PRECISION for pdsygst.

Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this routine. At present, *scale* is always returned as 1.0, it is returned here to allow for future enhancement.

info — (global) INTEGER.
If *info* = 0, the execution is successful.
If *info* <0, if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i*100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?hegst

Reduces a Hermitian-definite generalized eigenvalue problem to the standard form.

Syntax

```
call pchegst ( ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb,
    scale, info )
call pzhegst ( ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb,
    scale, info )
```

Description

This routine reduces complex Hermitian-definite generalized eigenproblems to the standard form.

In the following sub(*A*) denotes *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) and sub(*B*) denotes *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*n*-1).

If *ibtype* = 1, the problem is

$$\mathrm{sub}(A)x = \lambda \mathrm{sub}(B)x,$$

and sub(*A*) is overwritten by $\mathrm{inv}(U^H)\,\mathrm{sub}(A)\mathrm{inv}(U)$ or $\mathrm{inv}(L)\mathrm{sub}(A)\mathrm{inv}(L^H)$.

If *ibtype* = 2 or 3, the problem is

sub(A)sub(B)x = λx or sub(B)sub(A)x = λx,

and sub(A) is overwritten by Usub(A)U^H or L^Hsub(A)L.

sub(B) must have been previously factorized as U^HU or $LL^H$by `p?potrf`.

Input Parameters

ibtype (global) `INTEGER`. Must be 1 or 2 or 3.
If *itype* = 1, compute inv(U^H)sub(A)inv(U) or inv(L)sub(A)inv(L^H);
If *itype* = 2 or 3, compute Usub(A)U^H or
L^Hsub(A)L.

uplo (global)
`CHARACTER`. Must be `'U'` or `'L'`.
If *uplo* = `'U'`, the upper triangle of sub(A) is stored and sub (B) is factored as U^HU.
If *uplo* = `'L'`, the lower triangle of sub (A) is stored and sub (B) is factored as LL^H.

n (global) `INTEGER`. The order of the matrices sub (A) and sub (B) ($n \geq 0$).

a (local)
`COMPLEX` for `pchegst`
`DOUBLE COMPLEX` for `pzhegst`.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the array contains the local pieces of the *n*-by-*n* Hermitian distributed matrix sub(A). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(A) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(A) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

ia, ja (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix A, respectively.

desca (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix A.

b (local)
`COMPLEX` for `pchegst`
`DOUBLE COMPLEX` for `pzhegst`.

Pointer into the local memory to an array of dimension (*lld_b*, *LOCc*(*jb*+*n*-1)). On entry, the array contains the local pieces of the triangular factor from the Cholesky factorization of sub (*B*) as returned by `p?potrf`.

ib, jb (global) `INTEGER`. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.

descb (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

Output Parameters

a On exit, if *info* = 0, the transformed matrix, stored in the same format as sub(*A*).

scale (global)
`REAL` for `pchegst`
`DOUBLE PRECISION` for `pzhegst`.

Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this routine. At present, *scale* is always returned as 1.0, it is returned here to allow for future enhancement.

info (global) `INTEGER`.
If *info* = 0, the execution is successful.
If *info* <0, if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i*100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Driver Routines

Table 6-8 lists ScaLAPACK driver routines available for solving systems of linear equations, linear least-squares problems, standard eigenvalue and singular value problems, and generalized symmetric definite eigenproblems.

Table 6-8 ScaLAPACK Driver Routines

Type of Problem	Matrix type, storage scheme	Driver
Linear equations	general (partial pivoting)	`p?gesv` (simple driver) `p?gesvx` (expert driver)
	general band (partial pivoting)	`p?gbsv` (simple driver)
	general band (no pivoting)	`p?dbsv` (simple driver)
	general tridiagonal (no pivoting)	`p?dtsv` (simple driver)
	symmetric/Hermitian positive-definite	`p?posv` (simple driver) `p?posvx` (expert driver)
	symmetric/Hermitian positive-definite, band	`p?pbsv` (simple driver)
	symmetric/Hermitian positive-definite, tridiagonal	`p?ptsv` (simple driver)
Linear least squares problem	general *m*-by-*n*	`p?gels`
Symmetric eigenvalue problem	symmetric/Hermitian	`p?syev` (simple driver) `p?syevx` / `p?heevx` (expert driver)
Singular value decomposition	general *m*-by-*n*	`p?gesvd`
Generalized symmetric definite eigenvalue problem	symmetric/Hermitian, one matrix also positive-definite	`p?sygvx` / `p?hegvx` (expert driver)

p?gesv

Computes the solution to the system of linear equations with a square distributed matrix and multiple right-hand sides.

Syntax

```
call psgesv (n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pdgesv (n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pcgesv (n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pzgesv (n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
```

Description

The routine `p?gesv` computes the solution to a real or complex system of linear equations sub(A)* X = sub(B), where sub(A) = A(*ia:ia+n-1*, *ja:ja+n-1*) is an *n*-by-*n* distributed matrix and X and sub(B) = B(*ib:ib+n-1*, *jb:jb+nrhs-1*) are *n*-by-*nrhs* distributed matrices.

The LU decomposition with partial pivoting and row interchanges is used to factor sub(A) as sub(A) = $P\ L\ U$, where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. L and U are stored in sub(A). The factored form of sub(A) is then used to solve the system of equations sub(A)* X = sub(B).

Input Parameters

n — (global) `INTEGER`. The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(A) ($n \geq 0$).

nrhs — (global) `INTEGER`. The number of right hand sides, that is, the number of columns of the distributed submatrices B and X ($nrhs \geq 0$).

a,b — (local)
`REAL` for `psgesv`
`DOUBLE PRECISION` for `pdgesv`
`COMPLEX` for `pcgesv`
`DOUBLE COMPLEX` for `pzgesv`.
Pointers into the local memory to arrays of local dimension *a(lld_a, LOC_c(ja+n-1))* and *b(lld_b, LOC_c(jb+nrhs-1))*, respectively.

On entry, the array *a* contains the local pieces of the *n*-by-*n* distributed matrix sub(A) to be factored.

On entry, the array *b* contains the right hand side distributed matrix sub(*B*).

ia,ja (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of sub(*A*), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

ib,jb (global) INTEGER. The row and column indices in the global array *B* indicating the first row and the first column of sub(*B*), respectively.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

Output Parameters

a Overwritten by the factors *L* and *U* from the factorization sub(*A*) = *P L U*; the unit diagonal elements of *L* are not stored .

b Overwritten by the solution distributed matrix *X*.

ipiv (local) INTEGER array.
The dimension of *ipiv* is (LOC_r(*m_a*)+ *mb_a*).
This array contains the pivoting information. The (local) row i of the matrix was interchanged with the (global) row *ipiv*(i).
This array is tied to the distributed matrix *A*.

info (global) INTEGER. If *info*=0, the execution is successful.

info < 0:
if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.

info > 0:
If *info* = *k*, *U*(*ia+k-1*,*ja+k-1*) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution could not be computed.

p?gesvx

Uses the LU factorization to compute the solution to the system of linear equations with a square matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

```
call psgesvx (fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf,
    descaf, ipiv, equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond,
    ferr, berr, work, lwork, iwork, liwork, info)
call pdgesvx (fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf,
    descaf, ipiv, equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond,
    ferr, berr, work, lwork, iwork, liwork, info)
call pcgesvx (fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf,
    descaf, ipiv, equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond,
    ferr, berr, work, lwork, rwork, lrwork, info)
call pzgesvx (fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf,
    descaf, ipiv, equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond,
    ferr, berr, work, lwork, rwork, lrwork, info)
```

Description

This routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations $AX = B$, where A denotes the `n`-by-`n` submatrix A(`ia:ia+n-1`, `ja:ja+n-1`), B denotes the `n`-by-`nrhs` submatrix B(`ib:ib+n-1`, `jb:jb+nrhs-1`) and X denotes the `n`-by-`nrhs` submatrix X(`ix:ix+n-1`, `jx:jx+nrhs-1`).

Error bounds on the solution and a condition estimate are also provided.

In the following description, `af` stands for the subarray `af`(`iaf:iaf+n-1`, `jaf:jaf+n-1`).

The routine `p?gesvx` performs the following steps:

1. If `fact` = 'E', real scaling factors `R` and `C` are computed to equilibrate the system:

`trans` = 'N': $\text{diag}(R)*A*\text{diag}(C)*\text{diag}(C)^{-1}*X = \text{diag}(R)*B$

`trans` = 'T': $(\text{diag}(R)*A*\text{diag}(C))^{T}*\text{diag}(R)^{-1}*X = \text{diag}(C)*B$

trans = 'C': $(\text{diag}(R)*A*\text{diag}(C))^H*\text{diag}(R)^{-1}*X = \text{diag}(C)*B$

Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by diag(*R*)**A**diag(*C*) and *B* by diag(*R*)**B* (if *trans*='N') or diag(*c*)**B* (if *trans* = 'T' or 'C').

2. If *fact* = 'N' or 'E', the *LU* decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as *A* = *P L U*, where *P* is a permutation matrix, *L* is a unit lower triangular matrix, and *U* is upper triangular.

3. The factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than relative machine precision, steps 4 - 6 are skipped.

4. The system of equations is solved for *X* using the factored form of *A*.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix *X* is premultiplied by diag(*C*) (if *trans* = 'N') or diag(*R*) (if *trans* = 'T' or 'C') so that it solves the original system before equilibration.

Input Parameters

fact — (global) CHARACTER*1. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.

If *fact* = 'F' then, on entry, *af* and *ipiv* contain the factored form of *A*. If *equed* is not 'N', the matrix *A* has been equilibrated with scaling factors given by *r* and *c*. Arrays *a*, *af*, and *ipiv* are not modified.

If *fact* = 'N', the matrix *A* will be copied to *af* and factored.
If *fact* = 'E', the matrix *A* will be equilibrated if necessary, then copied to *af* and factored.

trans — (global) CHARACTER*1. Must be 'N', 'T', or 'C'.

Specifies the form of the system of equations:

If *trans* = 'N', the system has the form $A X = B$ (No transpose);
If *trans* = 'T', the system has the form $A^T X = B$ (Transpose);
If *trans* = 'C', the system has the form $A^H X = B$ (Conjugate transpose);

n — (global) INTEGER. The number of linear equations; the order of the submatrix *A* ($n \ge 0$).

nrhs (global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrices *B* and *X* (*nrhs* ≥ 0).

a,af,b,work (local)
REAL for psgesvx
DOUBLE PRECISION for pdgesvx
COMPLEX for pcgesvx
DOUBLE COMPLEX for pzgesvx.
Pointers into the local memory to arrays of local dimension a(*lld_a*, LOC_c(*ja*+*n*-1)), af(*lld_af*, LOC_c(*ja*+*n*-1)), b(*lld_b*, LOC_c(*jb*+*nrhs*-1)), *work(lwork)*, respectively.

The array *a* contains the matrix *A*. If *fact* = 'F' and *equed* is not 'N', then *A* must have been equilibrated by the scaling factors in *r* and/or *c*.

The array *af* is an input argument if *fact* = 'F' . In this case it contains on entry the factored form of the matrix *A*, i.e., the factors *L* and *U* from the factorization *A* = *P L U* as computed by p?getrf. If *equed* is not 'N', then *af* is the factored form of the equilibrated matrix *A*.

The array *b* contains on entry the matrix *B* whose columns are the right-hand sides for the systems of equations.

work(*) is a workspace array.
The dimension of *work* is (*lwork*).

ia,ja (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix *A*(*ia:ia*+*n*-1, *ja:ja*+*n*-1), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

iaf,jaf (global) INTEGER. The row and column indices in the global array *af* indicating the first row and the first column of the subarray af(*iaf:iaf*+*n*-1, *jaf:jaf*+*n*-1), respectively.

descaf (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *AF*.

ib,jb (global) INTEGER. The row and column indices in the global array B indicating the first row and the first column of the submatrix *B*(*ib:ib*+*n*-1, *jb:jb*+*nrhs*-1), respectively.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

ipiv (local) INTEGER array.
The dimension of *ipiv* is (LOC_r(*m_a*)+ *mb_a*).
The array *ipiv* is an input argument if *fact* = 'F'.
On entry, it contains the pivot indices from the factorization $A = P\ L\ U$ as computed by p?getrf; (local) row i of the matrix was interchanged with the (global) row *ipiv*(i).
This array must be aligned with *A*(*ia*:*ia*+*n*-*1*, *).

equed (global) CHARACTER*1. Must be 'N', 'R', 'C', or 'B'.
equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:
If *equed* = 'N', no equilibration was done (always true if *fact* = 'N');
If *equed* = 'R', row equilibration was done, that is, *A* has been premultiplied by diag(*r*);
If *equed* = 'C', column equilibration was done, that is, *A* has been postmultiplied by diag(*c*);
If *equed* = 'B', both row and column equilibration was done; *A* has been replaced by diag(*r*)**A**diag(*c*).

r, c (local) REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
Arrays, dimension LOC_r(m_a)and LOC_c(n_a), respectively.
The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*. These arrays are input arguments if *fact* = 'F' only; otherwise they are output arguments.
If *equed* = 'R' or 'B', *A* is multiplied on the left by diag(*r*); if *equed* = 'N' or 'C', *r* is not accessed.
If *fact* = 'F' and *equed* = 'R' or 'B', each element of *r* must be positive.

If *equed* = 'C' or 'B', *A* is multiplied on the right by diag(*c*); if *equed* = 'N' or 'R', *c* is not accessed.
If *fact* = 'F' and *equed* = 'C' or 'B', each element of *c* must be positive.
Array *r* is replicated in every process column, and is aligned with the distributed matrix *A*.
Array *c* is replicated in every process row, and is aligned with the distributed matrix *A*.

ix, jx (global) INTEGER. The row and column indices in the global array *X* indicating the first row and the first column of the submatrix *X*(*ix*:*ix*+*n*-*1*, *jx*:*jx*+*nrhs*-*1*), respectively.

descx (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *X*.

lwork (local or global) INTEGER. The dimension of the array *work* ; must be at least max(p?gecon(*lwork*), p?gerfs(*lwork*)) + LOC_r(n_a) .

iwork (local, psgesvx/pdgesvx only) INTEGER. Workspace array. The dimension of *iwork* is (*liwork*).

liwork (local, psgesvx/pdgesvx only) INTEGER. The dimension of the array *iwork* , must be at least LOC_r(n_a) .

rwork (local) REAL for pcgesvx;
DOUBLE PRECISION for pzgesvx.
Workspace array, used in complex flavors only.
The dimension of *rwork* is (*lrwork*).

lrwork (local or global, pcgesvx/pzgesvx only) INTEGER. The dimension of the array *rwork* ; must be at least 2*LOC_c(n_a) .

Output Parameters

x (local)
REAL for psgesvx
DOUBLE PRECISION for pdgesvx
COMPLEX for pcgesvx
DOUBLE COMPLEX for pzgesvx.
Pointer into the local memory to an array of local dimension x(lld_x, LOC_c(*jx+nrhs-1*)) .

If *info* = 0 , the array x contains the solution matrix *X* to the *original* system of equations. Note that *A* and *B* are modified on exit if *equed* ≠ 'N', and the solution to the *equilibrated* system is:
diag(*C*)$^{-1}$*X, if *trans* = 'N' and *equed* = 'C' or 'B'; and diag(*R*)$^{-1}$*X, if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'.

a Array *a* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'.
If *equed* ≠ 'N', *A* is scaled on exit as follows:
equed = 'R': *A* = diag(*R*)**A*
equed = 'C': *A* = *A**diag(*c*)
equed = 'B': *A* = diag(*R*)**A**diag(*c*)

af — If *fact* = 'N' or 'E', then *af* is an output argument and on exit returns the factors *L* and *U* from the factorization *A* = *P L U* of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *a* for the form of the equilibrated matrix.

b — Overwritten by diag(*R*)* *B* if *trans* = 'N' and *equed* = 'R' or 'B';
overwritten by diag(*c*)**B* if *trans* = 'T' and *equed* = 'C' or 'B';
not changed if *equed* = 'N'.

r, c — These arrays are output arguments if *fact* ≠ 'F'.
See the description of *r, c* in *Input Arguments* section.

rcond — (global) REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0,
for the working precision, the matrix may be poorly conditioned or even singular.

ferr, berr — (local) REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION LOC_c(n_b) each. Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

Arrays *ferr* and *berr* are both replicated in every process row, and are aligned with the matrices *B* and *X*.

ipiv — If *fact* = 'N' or 'E', then *ipiv* is an output argument and on exit contains the pivot indices from the factorization *A* = *P L U* of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E').

equed — If *fact* ≠ 'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

work(1) — If *info*=0, on exit *work*(1) returns the minimum value of *lwork* required for optimum performance.

iwork(1) — If *info*=0, on exit *iwork*(1) returns the minimum value of *liwork* required for optimum performance.

rwork(1) — If *info*=0, on exit *rwork*(1) returns the minimum value of *lrwork* required for optimum performance.

info INTEGER. If *info*=0, the execution is successful.

info < 0: if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.
If *info* = *i*, and *i* ≤ *n*, then *U*(*i*,*i*) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed.
If *info* = *i*, and *i* = *n* +1, then *U* is nonsingular, but *rcond* is less than machine precision. The factorization has been completed, but the matrix is singular to working precision and the solution and error bounds have not been computed.

p?gbsv

Computes the solution to the system of linear equations with a general banded distributed matrix and multiple right-hand sides.

Syntax

```
call psgbsv (n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work,
    lwork, info)
call pdgbsv (n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work,
    lwork, info)
call pcgbsv (n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work,
    lwork, info)
call pzgbsv (n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work,
    lwork, info)
```

Description

The routine p?gbsv computes the solution to a real or complex system of linear equations

$$\mathrm{sub}(A)*\ X = \mathrm{sub}(B),$$

where sub(*A*) = *A*(*1:n*, *ja:ja+n-1*) is an *n*-by-*n* real/complex general banded distributed matrix with *bwl* subdiagonals and *bwu* superdiagonals, and *X* and sub(*B*) = *B*(*ib:ib+n-1*, *1:nrhs*) are *n*-by-*nrhs* distributed matrices.

The *LU* decomposition with partial pivoting and row interchanges is used to factor sub(*A*) as sub(*A*) = *P L U Q*, where *P* and *Q* are permutation matrices, and *L* and *U* are banded lower and upper triangular matrices, respectively. The matrix *Q* represents reordering of columns for the sake of parallelism, while *P* represents reordering of rows for numerical stability using classic partial pivoting.

Input Parameters

n — (global) `INTEGER`. The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(*A*) ($n \geq 0$).

bwl — (global) `INTEGER`. The number of subdiagonals within the band of *A* ($0 \leq bwl \leq n-1$).

bwu — (global) `INTEGER`. The number of superdiagonals within the band of *A* ($0 \leq bwu \leq n-1$).

nrhs — (global) `INTEGER`. The number of right hand sides; the number of columns of the distributed submatrix sub(*B*) ($nrhs \geq 0$).

a, b — (local)

`REAL` for `psgbsv`
`DOUBLE PRECISION` for `pdgbsv`
`COMPLEX` for `pcgbsv`
`DOUBLE COMPLEX` for `pzgbsv`.

Pointers into the local memory to arrays of local dimension *a*(*lld_a*, LOC_c(*ja*+*n*-1)) and *b*(*lld_b*, LOC_c(*nrhs*)), respectively.

On entry, the array *a* contains the local pieces of the global array *A*.

On entry, the array *b* contains the right hand side distributed matrix sub(*B*).

ja — (global) `INTEGER`. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca — (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7;
else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

ib — (global) `INTEGER`. The row index in the global array *B* that points to the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

If *descb*(*dtype_*) = 502, then *dlen_* ≥ 7;
else if *descb*(*dtype_*) = 1, then *dlen_* ≥ 9.

work (local)
REAL for psgbsv
DOUBLE PRECISION for pdgbsv
COMPLEX for pcgbsv
DOUBLE COMPLEX for pzgbsv.

Workspace array of dimension (*lwork*).

lwork (local or global) INTEGER. The size of the array *work*, must be at least
lwork ≥ (NB+*bwu*)*(*bwl*+*bwu*)+6*(*bwl*+*bwu*)*(*bwl*+2*bwu*) +
+ max(*nrhs* *(NB+2*bwl*+4*bwu*), 1).

Output Parameters

a On exit, contains details of the factorization.
Note that the resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

b On exit, this array contains the local pieces of the solution distributed matrix *X*.

ipiv (local) INTEGER array.
The dimension of *ipiv* must be at least *desca*(NB).
This array contains pivot indices for local factorizations. You should not alter the contents between factorization and solve.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info INTEGER. If *info*=0, the execution is successful.
info < 0:

if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.

info > 0:
If *info* = k ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not nonsingular, and the factorization was not completed.

If `info` = k>NPROCS, the submatrix stored on processor `info`-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?dbsv

Solves a general band system of linear equations.

Syntax

```
call psdbsv (n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork,
    info)
call pddbsv (n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork,
    info)
call pcdbsv (n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork,
    info)
call pzdbsv (n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork,
    info)
```

Description

This routine solves the system of linear equations

$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs)$

where $A(1:n, ja:ja+n-1)$ is an n-by-n real/complex banded diagonally dominant-like distributed matrix with bandwidth `bwl`, `bwu`.

Gaussian elimination without pivoting is used to factor a reordering of the matrix into L U.

Input Parameters

n	(global) INTEGER. The order of the distributed submatrix A; ($n \geq 0$).
bwl	(global) INTEGER. Number of subdiagonals. $0 \leq bwl \leq n-1$.
bwu	(global) INTEGER. Number of subdiagonals. $0 \leq bwu \leq n-1$.
nrhs	(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix B ($nrhs \geq 0$).

a (local).
REAL for psdbsv
DOUBLE PRECISION for pddbsv
COMPLEX for pcdbsv
DOUBLE COMPLEX for pzdbsv.
Pointer into the local memory to an array with first dimension *lld_a* ≥(*bwl*+*bwu*+1) (stored in *desca*). On entry, this array contains the local pieces of the distributed matrix.

ja (global) INTEGER. The index in the global array *a* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array of dimension *dlen*.
if 1*d* type (*dtype_a*=501 or 502), *dlen* ≥ 7;
if 2*d* type (*dtype_a*=1), *dlen* ≥ 9.
The array descriptor for the distributed matrix *A*. Contains information of mapping of *A* to memory.

b (local)
REAL for psdbsv
DOUBLE PRECISION for pddbsv
COMPLEX for pcdbsv
DOUBLE COMPLEX for pzdbsv.
Pointer into the local memory to an array of local lead dimension *lld_b*≥NB. On entry, this array contains the local pieces of the right hand sides *B*(*ib*:*ib*+*n*-1, 1:*nrhs*).

ib (global) INTEGER. The row index in the global array *b* that points to the first row of the matrix to be operated on (which may be either all of *b* or a submatrix of *B*).

desb (global and local) INTEGER array of dimension *dlen*.
if 1*d* type (*dtype_b* =502), *dlen* ≥7;
if 2*d* type (*dtype_b* =1), *dlen* ≥ 9.
The array descriptor for the distributed matrix *B*. Contains information of mapping of *B* to memory.

work (local).

REAL for psdbsv
DOUBLE PRECISION for pddbsv
COMPLEX for pcdbsv

DOUBLE COMPLEX for pzdbsv.
Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.

lwork — (local or global) INTEGER.
Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.
lwork ≥ NB (*bwl*+*bwu*)+6 max(*bwl*,*bwu*)*max(*bwl*,*bwu*)
+max((max(*bwl*,*bwu*)*nrhs*), max(*bwl*,*bwu*)max(*bwl*,*bwu*))

Output Parameters

a — On exit, this array contains information containing details of the factorization. Note that permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.

b — On exit, this contains the local piece of the solutions distributed matrix *X*.

work — On exit, *work*(1) contains the minimal *lwork*.

info — (local) INTEGER. If *info*=0, the execution is successful.
< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
> 0: If *info* = *k* ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.
If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

p?dtsv

Solves a general tridiagonal system of linear equations.

Syntax

```
call psdtsv (n, nrhs, dl, d, du, ja, desca, b, ib, descb, work,
    lwork,info
call pddtsv (n, nrhs, dl, d, du, ja, desca, b, ib, descb, work,
    lwork,info
```

```
call pcdtsv (n, nrhs, dl, d, du, ja, desca, b, ib, descb, work,
    lwork,info
call pzdtsv (n, nrhs, dl, d, du, ja, desca, b, ib, descb, work,
    lwork,info
```

Description

This routine solves a system of linear equations

$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs)$

where $A(1:n, ja:ja+n-1)$ is an *n*-by-*n* complex tridiagonal diagonally dominant-like distributed matrix.

Gaussian elimination without pivoting is used to factor a reordering of the matrix into *L U*.

Input Parameters

n — (global) `INTEGER`. The order of the distributed submatrix *A* ($n \geq 0$).

nrhs — `INTEGER`. The number of right hand sides; the number of columns of the distributed matrix *B* ($nrhs \geq 0$).

dl — (local).
`REAL` for `psdtsv`
`DOUBLE PRECISION` for `pddtsv`
`COMPLEX` for `pcdtsv`
`DOUBLE COMPLEX` for `pzdtsv`.

Pointer to local part of global vector storing the lower diagonal of the matrix. Globally, *dl*(1) is not referenced, and *dl* must be aligned with *d*. Must be of size ≥ *desca*(*nb_*).

d — (local).
`REAL` for `psdtsv`
`DOUBLE PRECISION` for `pddtsv`
`COMPLEX` for `pcdtsv`
`DOUBLE COMPLEX` for `pzdtsv`.
Pointer to local part of global vector storing the main diagonal of the matrix.

du — (local).
`REAL` for `psdtsv`
`DOUBLE PRECISION` for `pddtsv`
`COMPLEX` for `pcdtsv`

DOUBLE COMPLEX for pzdtsv.
Pointer to local part of global vector storing the upper diagonal of the matrix. Globally, *du*(*n*) is not referenced, and *du* must be aligned with *d*.

ja (global) INTEGER. The index in the global array *a* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array of dimension *dlen*.
if 1*d* type (*dtype_a*=501 or 502), *dlen* >= 7;
if 2*d* type (*dtype_a*=1), *dlen* >= 9.
The array descriptor for the distributed matrix *A*. Contains information of mapping of *A* to memory.

b (local)
REAL for psdtsv
DOUBLE PRECISION for pddtsv
COMPLEX for pcdtsv
DOUBLE COMPLEX for pzdtsv.
Pointer into the local memory to an array of local lead dimension $lld\_b \geq$ NB.
On entry, this array contains the local pieces of the right hand sides *B*(*ib*:*ib*+*n*-1, 1:*nrhs*).

ib (global) INTEGER. The row index in the global array *b* that points to the first row of the matrix to be operated on (which may be either all of *b* or a submatrix of *B*).

desb (global and local) INTEGER array of dimension *dlen*.
if 1*d* type (*dtype_b* =502), $dlen \geq 7$;
if 2*d* type (*dtype_b* =1), $dlen \geq 9$.
The array descriptor for the distributed matrix *B*. Contains information of mapping of *B* to memory.

work (local).

REAL for psdtsv
DOUBLE PRECISION for pddtsv
COMPLEX for pcdtsv
DOUBLE COMPLEX for pzdtsv.
Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.

lwork (local or global) INTEGER.
Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.
lwork ≥ (12*NPCOL+3*NB)+max((10+2*min(100,*nrhs*))*NPCOL+4*nrhs*, 8*NPCOL).

Output Parameters

dl On exit, this array contains information containing the * factors of the matrix.

d On exit, this array contains information containing the * factors of the matrix. Must be of size ≥ *desca*(*nb_*).

du On exit, this array contains information containing the * factors of the matrix. Must be of size ≥ *desca*(*nb_*).

b On exit, this contains the local piece of the solutions distributed matrix *X*.

work On exit, *work*(1) contains the minimal *lwork*.

info (local) INTEGER. If *info*=0, the execution is successful.
< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
> 0: If *info* = *k* ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.
If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

p?posv

Solves a symmetric positive definite system of linear equations.

Syntax

```
call psposv (uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pdposv (uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pcposv (uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
```

```
call pzposv (uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
```

Description

This routine computes the solution to a real/complex system of linear equations

sub(*A*) * *X* = sub(*B*),

where sub(*A*) denotes *A*(*ia*:*ia*+*n*-1,*ja*:*ja*+*n*-1) and is an *n*-by-*n* symmetric/Hermitian distributed positive definite matrix and *X* and sub(*B*) denoting *B*(*ib*:*ib*+*n*-1,*jb*:*jb*+*nrhs*-1) are *n*-by-*nrhs* distributed matrices. The Cholesky decomposition is used to factor sub(*A*) as

sub(*A*) = U^T * *U*, if *uplo* = '*U*', or

sub(*A*) = *L* * *LT*, if *uplo* = '*L*',

where *U* is an upper triangular matrix and *L* is a lower triangular matrix. The factored form of sub(*A*) is then used to solve the system of equations.

Input Parameters

uplo (global).CHARACTER. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of sub(*A)* is stored.

n (global) INTEGER. The order of the distributed submatrix sub(*A)* (*n* ≥ 0).

nrhs INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix sub(*B)* (*nrhs* ≥ 0).

a (local)
REAL for psposv
DOUBLE PRECISION for pdposv
COMPLEX for pcposv
COMPLEX*16 for pzposv.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, this array contains the local pieces of the n-by-n symmetric distributed matrix sub(*A*) to be factored. If *uplo* = '*U*', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.If *uplo* = '*L*', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced.

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

b (local)
REAL for psposv
DOUBLE PRECISION for pdposv
COMPLEX for pcposv
COMPLEX*16 for pzposv.
Pointer into the local memory to an array of dimension (*lld_b*,*LOC*(*jb*+*nrhs*-1)). On entry, the local pieces of the right hand sides distributed matrix sub(*B*).

ib, jb (global) INTEGER. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

Output Parameters

a On exit, if *info* = 0, this array contains the local pieces of the factor *U* or *L* from the Cholesky factorization sub(*A*) = $U^H U$ or LL^H.

b On exit, if *info* = 0, sub (*B*) is overwritten by the solution distributed matrix *X*.

info (global) INTEGER.
If *info* =0, the execution is successful.
If *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
If *info* > 0: If *info* = *k*, the leading minor of order *k*, *A*(*ia*:*ia*+*k*-1,*ja*:*ja*+*k*-1) is not positive definite, and the factorization could not be completed, and the solution has not been computed.

p?posvx

Solves a symmetric or Hermitian positive definite system of linear equations.

Syntax

```
call psposvx (fact, uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf,
    descaf, equed, sr, sc, b, ib, jb, descb, x, ix, jx, descx, rcond,
    ferr, berr, work, lwork, iwork, liwork, info)
call pdposvx (fact, uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf,
    descaf, equed, sr, sc, b, ib, jb, descb, x, ix, jx, descx, rcond,
    ferr, berr, work, lwork, iwork, liwork, info)
call pcposvx (fact, uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf,
    descaf, equed, sr, sc, b, ib, jb, descb, x, ix, jx, descx, rcond,
    ferr, berr, work, lwork, iwork, liwork, info)
call pzposvx (fact, uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf,
    descaf, equed, sr, sc, b, ib, jb, descb, x, ix, jx, descx, rcond,
    ferr, berr, work, lwork, iwork, liwork, info)
```

Description

This routine uses the Cholesky factorization $A=U^TU$ or $A=LL^T$ to compute the solution to a real or complex system of linear equations

$A(ia:ia+n-1,ja:ja+n-1) * X = B(ib:ib+n-1,jb:jb+nrhs-1)$,

where $A(ia:ia+n-1,ja:ja+n-1)$ is a n-by-n matrix and X and $B(ib:ib+n-1,jb:jb+nrhs-1)$ are n-by-$nrhs$ matrices.

Error bounds on the solution and a condition estimate are also provided.

In the following comments *y* denotes $Y(iy:iy+m-1,jy:jy+k-1)$ a m-by-k matrix where *y* can be *a*, *af*, *b* and *x*.

The routine `p?posvx` performs the following steps:

1. If *fact* = 'E', real scaling factors *s* are computed to equilibrate the system:

$$\mathrm{diag}(sr)*A*\mathrm{diag}(sc)*\mathrm{inv}(\mathrm{diag}(sc))*X = \mathrm{diag}(sr)*B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by diag(*sr*)**A**diag(*sc*) and *B* by diag(*sr*)**B* .

2. If *fact* = 'N' or 'E', the Cholesky decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as

$A = U^T U$, if *uplo* = 'U', or
$A = L L^T$, if *uplo* = 'L',
where *U* is an upper triangular matrix and *L* is a lower triangular matrix.

3. The factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, steps 4-6 are skipped

4. The system of equations is solved for *X* using the factored form of *A*.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix *X* is premultiplied by diag(*sr*) so that it solves the original system before equilibration.

Input Parameters

fact (global) CHARACTER. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.

If *fact* = 'F': on entry, *af* contains the factored form of *A*. If *equed* = 'Y', the matrix *A* has been equilibrated with scaling factors given by *s*.
a and *af* will not be modified.

If *fact* = 'N', the matrix *A* will be copied to *af* and factored.
If *fact* = 'E', the matrix *A* will be equilibrated if necessary, then copied to *af* and factored.

uplo (global)
CHARACTER. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of *A* is stored.

n (global) INTEGER. The order of the distributed submatrix sub(*A*) ($n \geq 0$).

nrhs (global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrices *B* and *X*.($nrhs \geq 0$).

a (local)
REAL for psposvx
DOUBLE PRECISION for pdposvx

COMPLEX for pcposvx
DOUBLE COMPLEX for pzposvx.
Pointer into the local memory to an array of local dimension (*lld_a*, *LOCc*(*ja*+*n*-1)).On entry, the symmetric/Hermitian matrix *A*, except if *fact* = 'F' and *equed* = 'Y', then *A* must contain the equilibrated matrix diag(*sr*)**A**diag(*sc*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *A* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *A* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *A* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *A* is not referenced. *A* is not modified if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N' on exit.

ia, ja — (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

af — (local)
REAL for psposvx
DOUBLE PRECISION for pdposvx
COMPLEX for pcposvx
DOUBLE COMPLEX for pzposvx.
Pointer into the local memory to an array of local dimension (*lld_af*, *LOCc*(*ja*+*n*-1)).
If *fact* = 'F', then *af* is an input argument and on entry contains the triangular factor *U* or *L* from the Cholesky factorization $A = U^T*U$ or $A = L*L^T$, in the same storage format as *A*. If *equed .ne.* 'N', then *af* is the factored form of the equilibrated matrix diag(*sr*)**A**diag(*sc*).

iaf, jaf — (global) INTEGER. The row and column indices in the global array *af* indicating the first row and the first column of the submatrix *AF*, respectively.

descaf — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *AF*.

equed — (global).CHARACTER. Must be 'N' or 'Y'.
equed is an input argument if *fact* = 'F' . It specifies the form of equilibration that was done:
If *equed* = 'N', no equilibration was done (always true if *fact* = 'N');
If *equed* = 'Y', equilibration was done and *A* has been replaced by diag(*sr*)**A**diag(*sc*).

sr

(local)
REAL for psposvx
DOUBLE PRECISION for pdposvx
COMPLEX for pcposvx
DOUBLE COMPLEX for pzposvx.
Array, DIMENSION (*lld_a*).
The array *s* contains the scale factors for *A*. This array is an input argument if *fact* = 'F' only; otherwise it is an output argument.
If *equed* = 'N', *s* is not accessed.
If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.

b

(local)
REAL for psposvx
DOUBLE PRECISION for pdposvx
COMPLEX for pcposvx
DOUBLE COMPLEX for pzposvx.
Pointer into the local memory to an array of local dimension (*lld_b*, *LOCc*(*jb*+*nrhs*-1)).On entry, the *n*-by-*nrhs* right-hand side matrix *B*.

ib,jb

(global) INTEGER. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.

descb

(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

x

(local)
REAL for psposvx
DOUBLE PRECISION for pdposvx
COMPLEX for pcposvx
DOUBLE COMPLEX for pzposvx.
Pointer into the local memory to an array of local dimension (*lld_x*, *LOCc*(*jx*+*nrhs*-1)).

ix,jx

(global) INTEGER. The row and column indices in the global array *x* indicating the first row and the first column of the submatrix *X*, respectively.

descx

(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *X*.

work

(local)
REAL for psposvx
DOUBLE PRECISION for pdposvx

COMPLEX for pcposvx
DOUBLE COMPLEX for pzposvx.
Workspace array, DIMENSION (*lwork*);

lwork (local or global)
INTEGER.
The dimension of the array *work*. *lwork* is local input and must be at least
lwork = max(p?pocon(*lwork*), p?porfs(*lwork*)) + *LOCr*(*n_a*).
lwork = 3**desca*(*lld_*)

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

liwork (local or global)

INTEGER.
The dimension of the array *iwork*. *liwork* is local input and must be at least
liwork = *desca*(*lld_*) *liwork* = *LOCr*(*n_a*).
If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, if *fact* = 'E' and *equed* = 'Y', *a* is overwritten by diag(*sr*)**a**diag(*sc*).

af If *fact* = 'N', then *af* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A = U^T*U$ or $A = L*L^T$ of the original matrix *A*.
If *fact* = 'E', then *af* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A = U^T*U$ or $A = L*L^T$ of the equilibrated matrix *A* (see the description of *A* for the form of the equilibrated matrix).

equed If *fact* ≠ 'F' , then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

sr This array is an output argument if *fact* ≠ 'F' .
See the description of *sr* in *Input Arguments* section.

sc This array is an output argument if *fact* ≠ 'F' .
See the description of *sc* in *Input Arguments* section.

b On exit, if *equed* = 'N', *b* is not modified; if *trans* = 'N' and *equed* = 'R' or 'B', *b* is overwritten by diag(*r*)**b*; if *trans* = 'T' or 'C' and *equed* = 'C' or 'B', *b* is overwritten by diag(*c*)**b*.

x (local)
REAL for psposvx
DOUBLE PRECISION for pdposvx
COMPLEX for pcposvx
DOUBLE COMPLEX for pzposvx.

If *info* = 0 the *n*-by-*nrhs* solution matrix *X* to the original system of equations. Note that *A* and *B* are modified on exit if *equed .ne*. 'N', and the solution to the equilibrated system is inv(diag(*sc*))**X* if *trans* = 'N' and *equed* = 'C' or 'B', or inv(diag(*sr*))**X* if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'.

rcond (global)
REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(*LOC*,*n_b*). The estimated forward error bounds for each solution vector *X*(*j*) (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution, *ferr*(*j*) bounds the magnitude of the largest entry in (*X*(*j*) - *xtrue*) divided by the magnitude of the largest entry in *X*(*j*). The quality of the error bound depends on the quality of the estimate of norm(inv(*A*)) computed in the code; if the estimate of norm(inv(*A*)) is accurate, the error bound is guaranteed.

berr (local)
REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(*LOC*,*n_b*).
The componentwise relative backward error of each solution vector *X*(*j*) (the smallest relative change in any entry of *A* or *B* that makes *X*(*j*) an exact solution).

info (global) INTEGER.
If *info*=0, the execution is successful.
< 0: if *info* = -*i*, the *i*-th argument had an illegal value
> 0: if *info* = *i*, and *i* is <= *n*: if *info* = *i*, the leading minor of order *i* of *a* is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed.
= *n*+1: *rcond* is less than machine precision. The factorization has been completed, but the matrix is singular to working precision, and the solution and error bounds have not been computed.

p?pbsv

Solves a symmetric/Hermitian positive definite banded system of linear equations.

Syntax

```
call pspbsv (uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork,
    info)
call pdpbsv (uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork,
    info)
call pcpbsv (uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork,
    info)
call pzpbsv (uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork,
    info)
```

Description

This routine solves a system of linear equations

$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs)$

where $A(1:n, ja:ja+n-1)$ is an *n*-by-*n* real/complex banded symmetric positive definite distributed matrix with bandwidth *bw*.

Cholesky factorization is used to factor a reordering of the matrix into *L L'*.

Input Parameters

uplo (global) CHARACTER. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular of *A* is stored.

If *uplo* = 'U', the upper triangular *A* is stored
If *uplo* = 'L', the lower triangular of *A* is stored.

n (global) INTEGER. The order of the distributed matrix *A* ($n \geq 0$).

bw (global) INTEGER. The number of subdiagonals in *L or U.* $0 \leq bw \leq n$-1.

nrhs (global) INTEGER. The number of right-hand sides; the number of columns in *B* ($nrhs \geq 0$).

a (local).
REAL for pspbsv
DOUBLE PRECISION for pdpbsv
COMPLEX for pcpbsv
DOUBLE COMPLEX for pzpbsv.
Pointer into the local memory to an array with first dimension $lld\_a \geq (bw+1)$ (stored in *desca*).
On entry, this array contains the local pieces of the distributed matrix sub(*A*) to be factored.

ja (global) INTEGER. The index in the global array *a* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

b (local)
REAL for pspbsv
DOUBLE PRECISION for pdpbsv
COMPLEX for pcpbsv
DOUBLE COMPLEX for pzpbsv.
Pointer into the local memory to an array of local lead dimension $lld\_b \geq$ NB.
On entry, this array contains the local pieces of the right hand sides
B(*ib*:*ib*+*n*-1, 1:*nrhs*).

ib (global) INTEGER. The row index in the global array *b* that points to the first row of the matrix to be operated on (which may be either all of *b* or a submatrix of *B*).

desb (global and local) INTEGER array of dimension *dlen*.
if 1D type ($dtype\_b$ =502), $dlen \geq 7$;
if 2D type ($dtype\_b$ =1), $dlen \geq 9$.
The array descriptor for the distributed matrix *B*. Contains information of mapping of *B* to memory.

work (local).

REAL for pspbsv
DOUBLE PRECISION for pdpbsv
COMPLEX for pcpbsv
DOUBLE COMPLEX for pzpbsv.
Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.

lwork (local or global) INTEGER.
Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.
lwork ≥ (NB+2**bw*)**bw* +max((*bw***nrhs*), *bw***bw*)

Output Parameters

a On exit, this array contains information containing details of the factorization. Note that permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.

b On exit, contains the local piece of the solutions distributed matrix *X*.

work On exit, *work*(1) contains the minimal *lwork*.

info (global).
INTEGER. If *info*=0, the execution is successful.
< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
> 0: If *info* = *k* ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.
If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

p?ptsv

Solves a symmetric or Hermitian positive definite tridiagonal system of linear equations.

Syntax

```
call psptsv (n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
call pdptsv (n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
call pcptsv (n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
call pzptsv (n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
```

Description

This routine solves a system of linear equations

$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs)$

where $A(1:n, ja:ja+n-1)$ is an n-by-n real tridiagonal symmetric positive definite distributed matrix.

Cholesky factorization is used to factor a reordering of the matrix into $L L'$.

Input Parameters

n (global) `INTEGER`. The order of matrix A ($n \geq 0$).

nrhs (global) `INTEGER`. The number of right-hand sides; the number of columns of the distributed submatrix B ($nrhs \geq 0$).

d (local)
`REAL` for `psptsv`
`DOUBLE PRECISION` for `pdptsv`
`COMPLEX` for `pcptsv`
`DOUBLE COMPLEX` for `pzptsv`.
Pointer to local part of global vector storing the main diagonal of the matrix.

e (local)
`REAL` for `psptsv`
`DOUBLE PRECISION` for `pdptsv`
`COMPLEX` for `pcptsv`

	`DOUBLE COMPLEX` for `pzptsv`. Pointer to local part of global vector storing the upper diagonal of the matrix. Globally, *du*(n) is not referenced, and *du* must be aligned with *d*.
ja	(global) `INTEGER.` The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).
desca	(global and local) `INTEGER` array of dimension *dlen*. if 1*d* type (*dtype_a*=501 or 502), *dlen* >= 7; if 2*d* type (*dtype_a*=1), *dlen* >= 9. The array descriptor for the distributed matrix *A*. Contains information of mapping of *A* to memory.
b	(local) `REAL` for `psptsv` `DOUBLE PRECISION` for `pdptsv` `COMPLEX` for `pcptsv` `DOUBLE COMPLEX` for `pzptsv`. Pointer into the local memory to an array of local lead dimension *lld_b*≥`NB`. On entry, this array contains the local pieces of the right hand sides *B*(*ib*:*ib*+*n*-1, 1:*nrhs*).
ib	(global) `INTEGER`. The row index in the global array *b* that points to the first row of the matrix to be operated on (which may be either all of *b* or a submatrix of *B*).
desb	(global and local) `INTEGER` array of dimension *dlen*. if 1*d* type (*dtype_b* =502), *dlen* ≥7; if 2*d* type (*dtype_b* =1), *dlen* ≥ 9. The array descriptor for the distributed matrix *B*. Contains information of mapping of *B* to memory.
work	(local). `REAL` for `psptsv` `DOUBLE PRECISION` for `pdptsv` `COMPLEX` for `pcptsv` `DOUBLE COMPLEX` for `pzptsv`. Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.
lwork	(local or global) `INTEGER.` Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned. *lwork* ≥ (12*`NPCOL`+3*`NB`)+`max`((10+2*min(100,*nrhs*))*`NPCOL`+4**nrhs*, 8*`NPCOL`).

Output Parameters

d — On exit, this array contains information containing the factors of the matrix. Must be of size ≥ *desca*(*nb_*).

e — On exit, this array contains information containing the factors of the matrix. Must be of size ≥ *desca*(*nb_*).

b — On exit, this contains the local piece of the solutions distributed matrix *X*.

work — On exit, *work*(1) contains the minimal *lwork*.

info — (local) INTEGER. If *info*=0, the execution is successful.
< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
> 0: If *info* = *k* ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.
If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

p?gels

Solves overdetermined or underdetermined linear systems involving a matrix of full rank.

Syntax

```
call psgels ( trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb,
    work, lwork, info )
call pdgels ( trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb,
    work, lwork, info )
call pcgels ( trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb,
    work, lwork, info )
call pzgels ( trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb,
    work, lwork, info )
```

Description

This routine solves overdetermined or underdetermined real/ complex linear systems involving an *m*-by-*n* matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1), or its transpose/ conjugate-transpose, using a *QR* or *LQ* factorization of
sub(*A*). It is assumed that sub(*A*) has full rank.

The following options are provided:

1. If *trans* = 'N' and $m \geq n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

minimize $\|$ sub (*B*) -sub(*A*) *X* $\|$

2. If *trans* = 'N' and $m < n$: find the minimum norm solution of an underdetermined system sub(*A*) *X* = sub(*B*).

3. If *trans* = 'T' and $m \geq n$: find the minimum norm solution of an undetermined system sub$(A)^T X$ = sub(*B*).

4. If *trans* = 'T' and $m < n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

minimize $\|$ sub(*B*) - sub$(A)^T X$ $\|$

where sub(*B*) denotes *B*(*ib*:*ib*+*m*-1, *jb*:*jb*+*nrhs*-1) when *trans* = 'N' and *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*nrhs*-1) otherwise. Several right hand side vectors *b* and solution vectors *x* can be handled in a single call;
When when *trans* = 'N', the solution vectors are stored as the columns of the *n*-by-*nrhs* right hand side matrix sub(*B*) and the *m*-by-*nrhs* right hand side matrix sub(*B*) otherwise.

Input Parameters

trans	(global) CHARACTER. Must be 'N', or'T'. If *trans* = 'N', the linear system involves matrix sub(*A*); If *trans* = 'T', the linear system involves the transposed matrix A^T (for real flavors only).
m	(global) INTEGER. The number of rows in the distributed submatrix sub (*A*) ($m \geq 0$).
n	(global) INTEGER. The number of columns in the distributed submatrix sub (*A*) ($n \geq 0$).
nrhs	(global) INTEGER. The number of right-hand sides; the number of columns in the distributed submatrices sub(*B*) and *X*. ($nrhs \geq 0$).

a
(local)
REAL for psgels
DOUBLE PRECISION for pdgels
COMPLEX for pcgels
DOUBLE COMPLEX for pzgels.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, contains the *m*-by-*n* matrix *A*.

ia, *ja*
(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca
(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

b
(local)
REAL for psgels
DOUBLE PRECISION for pdgels
COMPLEX for pcgels
DOUBLE COMPLEX for pzgels.
Pointer into the local memory to an array of local dimension (*lld_b*, *LOCc*(*jb*+*nrhs*-1)). On entry, this array contains the local pieces of the distributed matrix *B* of right-hand side vectors, stored columnwise; sub(*B*) is *m*-by-*nrhs* if *trans*='N', and *n*-by-*nrhs* otherwise.

ib, *jb*
(global) INTEGER. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.

descb
(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

work
(local)
REAL for psgels
DOUBLE PRECISION for pdgels
COMPLEX for pcgels
DOUBLE COMPLEX for pzgels.
Workspace array with dimension *lwork*.

lwork
(local or global)
INTEGER.The dimension of the array *work*
lwork is local input and must be at least
lwork >= *ltau* + max(*lwf*, *lws*) where
if $m \ge n$, then
ltau = numroc(*ja*+min(*m*,*n*)-1, *nb_a*, MYCOL, *csrc_a*, NPCOL),
lwf = *nb_a* * (*mpa*0 + *nqa*0 + *nb_a*)

lws = max((*nb_a**(*nb_a*-1))/2, (*nrhsqb*0 + *mpb*0)**nb_a*) + *nb_a* * *nb_a*
else
ltau = numroc(*ia*+min(*m*,*n*)-1, *mb_a*, MYROW, *rsrc_a*, NPROW),
lwf = *mb_a* * (*mpa*0 + *nqa*0 + *mb_a*)
lws = max((*mb_a**(*mb_a*-1))/2, (*npb*0 + max(*nqa*0 + numroc(numroc(*n*+*iroffb*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nrhsqb*0))**mb_a*) + *mb_a* * *mb_a*

End if
where *lcmp* = *lcm* / NPROW with *lcm* = ilcm(NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*),
icoffa = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
iacol= indxg2p(*ja*, *nb_a*, MYROW, *rsrc_a*, NPROW)
*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),
iroffb = mod(*ib*-1, *mb_b*),
icoffb = mod(*jb*-1, *nb_b*),
ibrow = indxg2p(*ib*, *mb_b*, MYROW, *rsrc_b*, NPROW),
ibcol = indxg2p(*jb*, *nb_b*, MYCOL, *csrc_b*, NPCOL),
*mpb*0 = numroc(*m*+*iroffb*, *mb_b*, MYROW, *icrow*, NPROW),
*nqb*0 = numroc(*n*+*icoffb*, *nb_b*, MYCOL, *ibcol*, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — On exit, If $m \geq n$, sub(A) is overwritten by the details of its QR factorization as returned by p?geqrf; if $m < n$, sub(A) is overwritten by details of its LQ factorization as returned by p?gelqf.

b — On exit, sub(B) is overwritten by the solution vectors, stored columnwise: if *trans* = 'N' and $m \geq n$, rows 1 to n of sub(B) contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements n+1 to m in that column;

	if *trans* = 'N' and $m < n$, rows 1 to *n* of sub(*B*) contain the minimum norm solution vectors; if *trans* = 'T' and $m \geq n$, rows 1 to *m* of sub(*B*) contain the minimum norm solution vectors; if *trans* = 'T' and $m < n$, rows 1 to *m* of sub(*B*) contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements *m*+1 to *n* in that column.
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?syev

Computes selected eigenvalues and eigenvectors of a symmetric matrix.

Syntax

```
call pssyev ( jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work,
    lwork, info )
call pdsyev ( jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work,
    lwork, info )
```

Description

This routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A* by calling the recommended sequence of ScaLAPACK routines.
In its present form, the routine assumes a homogeneous system and makes no checks for consistency of the eigenvalues or eigenvectors across the different processes. Because of this, it is possible that a heterogeneous system may return incorrect results without any error messages.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

jobz (global).CHARACTER. Must be 'N' or 'V'.
Specifies if it is necessary to compute the eigenvectors:
If *jobz* ='N', then only eigenvalues are computed.
If *jobz* ='V', then eigenvalues and eigenvectors are computed.

uplo (global).CHARACTER. Must be 'U' or 'L'.
Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored:
If *uplo* = 'U', *a* stores the upper triangular part of *A*.
If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n (global) INTEGER. The number of rows and columns of the matrix *A* ($n \geq 0$).

a (local)
REAL for pssyev.
DOUBLE PRECISION for pdsyev.
Block cyclic array of global dimension (*n*,*n*) and local dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the symmetric matrix *A*. If *uplo* = 'U', only the upper triangular part of *A* is used to define the elements of the symmetric matrix. If *uplo* = 'L', only the lower triangular part of *A* is used to define the elements of the symmetric matrix.

ia,ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

iz,jz (global) INTEGER. The row and column indices in the global array *z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *Z*.

work (local)
REAL for pssyev.
DOUBLE PRECISION for pdsyev.
Array, DIMENSION (*lwork*).

lwork (local)
INTEGER. See below for definitions of variables used to define *lwork*.
If no eigenvectors are requested (*jobz* = 'N') then *lwork* ≥ 5**n* + *sizesytrd* + 1 where
sizesytrd = The workspace requirement for p?sytrd and is max(NB * (*np*

+1), 3 * NB).
If eigenvectors are requested (*jobz* = 'V') then the amount of workspace required to guarantee that all eigenvectors are computed is:
qrmem = 2**n*-2
lwmin = 5**n* + *n** *ldc* + max(*sizemqrleft*, *qrmem*) + 1

Variable definitions:
NB = *desca*(*mb_*) = *desca*(*nb_*) = * *descz*(*mb_*) = *descz*(*nb_*)
nn = max(*n*, NB, 2)
desca(*rsrc_*) = *desca*(*rsrc_*) = *descz*(*rsrc_*) = * *descz*(*csrc_*) = 0
np = numroc(*nn*, NB, 0, 0, NPROW)
nq = numroc(max(*n*, NB, 2), NB, 0, 0, NPCOL)
nrc = numroc(*n*, NB, *myprowc*, 0, NPROCS)
ldc = max(1, *nrc*)
sizemqrleft = The workspace requirement for p?ormtr when it's *side* argument is 'L'.
With *myprowc* defined when a new context is created as:
call blacs_get(*desca*(*ctxt_*), 0, *contextc*) call
blacs_gridinit(*contextc*, 'R', NPROCS, 1) call
blacs_gridinfo(*contextc*, *nprowc*, *npcolc*, *myprowc*, *mypcolc*)

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, the lower triangle (if *uplo*='L') or the upper triangle (if *uplo*='U') of *A*, including the diagonal, is destroyed.

w (global).
REAL for pssyev
DOUBLE PRECISION for pdsyev
Array, DIMENSION (*n*).
On normal exit, the first *m* entries contain the selected eigenvalues in ascending order.

z (local).
REAL for pssyev
DOUBLE PRECISION for pdsyev
Array, global dimension (*n*, *n*), local dimension (*lld_z*, *LOCc*(*jz*+*n*-1)).If

`jobz` = 'V', then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If *jobz* = 'N', then *z* is not referenced.

work(1) — On output, *work(1)* returns the workspace needed to guarantee completion. If the input parameters are incorrect, *work(1)* may also be incorrect.
If *jobz* = 'N' *work(1)* = minimal (optimal) amount of workspace
If *jobz* = 'V' *work(1)* = minimal workspace required to generate all the eigenvectors.

info — (global)
INTEGER.
If *info* = 0, the execution is successful.

If *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info* > 0:
If *info*= 1 through *n*, the *i*-th eigenvalue did not converge in ?steqr2 after a total of 30*n* iterations.
If *info*= *n*+1, then p?syev has detected heterogeneity by finding that eigenvalues were not identical across the process grid. In this case, the accuracy of the results from p?syev cannot be guaranteed.

p?syevx

Computes selected eigenvalues and, optionally, eigenvectors of a symmetric matrix.

Syntax

```
call pssyevx (jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu,
    abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, iwork, liwork,
    ifail, iclustr, gap, info)
call pdsyevx (jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu,
    abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, iwork, liwork,
    ifail, iclustr, gap, info)
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A* by calling the recommended sequence of ScaLAPACK routines. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

np = the number of rows local to a given process.
nq = the number of columns local to a given process.

jobz (global).`CHARACTER*1`. Must be `'N'` or `'V'`.
Specifies if it is necessary to compute the eigenvectors:
If *jobz* =`'N'`, then only eigenvalues are computed.
If *jobz* =`'V'`, then eigenvalues and eigenvectors are computed.

range (global).`CHARACTER*1`. Must be `'A'`, `'V'`, or `'I'`.
If *range* =`'A'`, all eigenvalues will be found.
If *range* =`'V'`, all eigenvalues in the half-open interval [*vl*, *vu*] will be found.
If *range* =`'I'`, the eigenvalues with indices *il* through *iu* will be found.

uplo (global).`CHARACTER*1`. Must be `'U'` or `'L'`.
Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored:
If *uplo* = `'U'`, *a* stores the upper triangular part of *A*.
If *uplo* = `'L'`, *a* stores the lower triangular part of *A*.

n (global) `INTEGER`. The number of rows and columns of the matrix *A* ($n \geq 0$).

a (local).
`REAL` for `pssyevx`
`DOUBLE PRECISION` for `pdsyevx`.
Block cyclic array of global dimension (*n*,*n*) and local dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the symmetric matrix *A*. If *uplo* = `'U'`, only the upper triangular part of *A* is used to define the elements of the symmetric matrix. If *uplo* = `'L'`, only the lower triangular part of *A* is used to define the elements of the symmetric matrix.

ia, ja (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

vl, vu

(global)
`REAL` for `pssyevx`
`DOUBLE PRECISION` for `pdsyevx`.
If *range* ='V', the lower and upper bounds of the interval to be searched for eigenvalues; *vl* ≤ *vu*.
Not referenced if *range* ='A' or 'I'.

il, iu

(global)
`INTEGER`. If *range* ='I', the indices of the smallest and largest eigenvalues to be returned.
Constraints:
il ≥ 1
min(*il*,*n*) ≤ *iu* ≤ *n*
Not referenced if *range* ='A' or 'V'.

abstol

(global).
`REAL` for `pssyevx`
`DOUBLE PRECISION` for `pdsyevx`.
If *jobz*='V', setting *abstol* to `p?lamch`(*context*, 'U') yields the most orthogonal eigenvectors.

The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*,*b*] of width less than or equal to
abstol + *eps* * max(|*a*|,|*b*|),
where *eps* is the machine precision. If *abstol* is less than or equal to zero, then *eps**norm(T) will be used in its place, where norm(T) is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*`p?lamch`('S') not zero.If this routine returns with ((mod(*info*,2).*ne*.0).*or*. * (mod(*info*/8,2).*ne*.0)), indicating that some eigenvalues or eigenvectors did not converge, try setting *abstol* to 2*`p?lamch`('S').

orfac

(global).
`REAL` for `pssyevx`
`DOUBLE PRECISION` for `pdsyevx`.
Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within *tol*=*orfac**norm(*A*) of each other are to be reorthogonalized. However, if the workspace is insufficient (see *lwork*), *tol* may be decreased until all eigenvectors to be reorthogonalized

can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of 10^3 is used if *orfac* is negative. *orfac* should be identical on all processes.

iz,jz (global) INTEGER. The row and column indices in the global array *z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *Z*.*descz*(*ctxt_*) must equal *desca*(*ctxt_*).

work (local)
REAL for pssyevx.
DOUBLE PRECISION for pdsyevx.
Array, DIMENSION (*lwork*).

lwork (local) INTEGER. The dimension of the array *work*.
See below for definitions of variables used to define *lwork*.
If no eigenvectors are requested (*jobz* = 'N') then *lwork* ≥ 5 * *n* + max(5 * *nn*, NB * (*np*0 + 1)).
If eigenvectors are requested (*jobz* = 'V') then the amount of workspace required to guarantee that all eigenvectors are computed is:

lwork >= 5**n* + max(5**nn*, *np*0 * *mq*0 + 2 * NB * NB) +iceil(*neig*, NPROW*NPCOL)**nn*

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following to *lwork*:
(*clustersize*-1)**n*
where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:
{*w*(*k*),...,*w*(*k*+*clustersize*-1) |
w(*j*+1) ≤ *w*(*j*)) + *orfac**2*norm(*A*)}

Variable definitions:
neig = number of eigenvectors requested
NB = *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*)
nn = max(*n*, NB, 2)
desca(*rsrc_*) = *desca*(*nb_*) = *descz*(*rsrc_*) = *descz*(*csrc_*) = 0
*np*0 = numroc(*nn*, NB, 0, 0, NPROW)
*mq*0 = numroc(max(*neig*, NB, 2), NB, 0, 0, NPCOL) iceil(*x*, *y*) is a ScaLAPACK function returning ceiling(*x*/*y*)

When *lwork* is too small:
If *lwork* is too small to guarantee orthogonality, p?syevx attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.
If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*=-23 is returned. Note that when *range*='V', p?syevx does not know how many eigenvectors are requested until the eigenvalues are computed. Therefore, when *range*='V' and as long as *lwork* is large enough to allow p?syevx to compute the eigenvalues, p?syevx will compute the eigenvalues and as many eigenvectors as it can.

Relationship between workspace, orthogonality & performance:
Greater performance can be achieved if adequate workspace is provided. On the other hand, in some situations, performance can decrease as the workspace provided increases above the workspace amount shown below:

For optimal performance, greater workspace may be needed, that is,
lwork ≥ max(*lwork*, 5**n* + *nsytrd_lwopt*)
Where:
lwork, as defined previously, depends upon the number of eigenvectors requested, and
nsytrd_lwopt = *n* + 2*(*anb*+1)*(4**nps*+2) + (*nps* + 3) * *nps*

anb = pjlaenv(*desca*(*ctxt_*), 3, 'p?syttrd', 'L', 0, 0, 0, 0)
sqnpc = int(*sqrt*(*dble*(NPROW * NPCOL)))
nps = max(numroc(*n*, 1, 0, 0, *sqnpc*), 2**anb*)

numroc is a ScaLAPACK tool functions;
pjlaenv is a ScaLAPACK environmental inquiry function
MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

For large *n*, no extra workspace is needed, however the biggest boost in performance comes for small *n*, so it is wise to provide the extra workspace (typically less than a Megabyte per process).

If *clustersize* ≥ *n*/*sqrt*(NPROW*NPCOL), then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. In the limit (that is, *clustersize* = *n*-1) p?stein will perform no better than ?stein on 1 processor.
For *clustersize* = *n*/*sqrt*(NPROW*NPCOL) reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.
For *clustersize* > *n*/*sqrt*(NPROW*NPCOL) execution time will grow as the

square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.

iwork (local) INTEGER. Workspace array.

liwork (local) INTEGER, dimension of *iwork*.
liwork ≥ 6 * *nnp*
Where: *nnp* = max(*n*, NPROW*NPCOL + 1, 4)
If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

m (global) INTEGER. The total number of eigenvalues found; $0 \le m \le n$.

w (global).
REAL for pssyevx
DOUBLE PRECISION for pdsyevx
Array, DIMENSION (*n*).
The first *m* elements contain the selected eigenvalues in ascending order.

z (local).
REAL for pssyevx
DOUBLE PRECISION for pdsyevx
Array, global dimension (*n*, *n*),
local dimension (*lld_z*, *LOCc*(*jz*+*n*-1))
If *jobz* ='V', then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
If *jobz* = 'N', then *z* is not referenced.

work(1) On exit, returns workspace adequate workspace to allow optimal performance.

iwork(1) On return, *iwork(1)* contains the amount of integer workspace required

ifail (global) INTEGER.Array, DIMENSION (*n*).
If *jobz* ='V', then on normal exit, the first *m* elements of *ifail* are zero. If (mod(*info*,2).*ne*.0) on exit, then *ifail* contains the indices of the eigenvectors that failed to converge.
If *jobz* = 'N', then *ifail* is not referenced.

iclustr (global) INTEGER.
Array, DIMENSION (2*NPROW*NPCOL)
This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*).Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2**i*-1) to *iclustr*(2**i*), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr*() is a zero terminated array. (*iclustr*(2**k*).*ne*.0.*and*. *iclustr*(2**k*+1).*eq*.0) if and only if *k* is the number of clusters.
iclustr is not referenced if *jobz* = 'N'

gap (global)
REAL for pssyevx
DOUBLE PRECISION for pdsyevx
Array, DIMENSION (NPROW*NPCOL)
This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the i^{th} cluster may be as high as (*C* * *n*) / *gap*(*i*) where *C* is a small constant.

info (global) INTEGER.
If *info* = 0, the execution is successful.
If *info*< 0:
If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
If *info* > 0: if (mod(*info*,2).*ne*.0), then one or more eigenvectors failed to converge. Their indices are stored in *ifail*. Ensure *abstol*=2.0*p?lamch('U')
if (mod(*info*/2,2).*ne*.0),then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient

workspace.The indices of the clusters are stored in the array *iclustr*.
if (mod(*info*/4,2).*ne*.0), then space limit prevented p?syevx from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.
if (mod(*info*/8,2).*ne*.0), then p?stebz failed to compute eigenvalues. Ensure *abstol*=2.0*p?lamch('U').

p?heevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

```
call pcheevx (jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu,
    abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, rwork, lrwork,
    iwork, liwork, ifail, iclustr, gap, info)
call pzheevx (jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu,
    abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, rwork, lrwork,
    iwork, liwork, ifail, iclustr, gap, info)
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A* by calling the recommended sequence of ScaLAPACK routines. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

np = the number of rows local to a given process.
nq = the number of columns local to a given process.

jobz (global).CHARACTER*1. Must be 'N' or 'V'.
Specifies if it is necessary to compute the eigenvectors:
If *jobz* ='N', then only eigenvalues are computed.
If *jobz* ='V', then eigenvalues and eigenvectors are computed.

range (global).CHARACTER*1. Must be 'A', 'V', or 'I'.
If *range* ='A', all eigenvalues will be found.
If *range* ='V', all eigenvalues in the half-open interval [*vl*, *vu*] will be found.
If *range* ='I', the eigenvalues with indices *il* through *iu* will be found.

uplo (global).CHARACTER*1. Must be 'U' or 'L'.
Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is stored:
If *uplo* = 'U', *a* stores the upper triangular part of *A*.
If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n (global) INTEGER. The number of rows and columns of the matrix *A* ($n \geq 0$).

a (local).
COMPLEX for pcheevx
DOUBLE COMPLEX for pzheevx.
Block cyclic array of global dimension (*n*,*n*) and local dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the Hermitian matrix *A*. If *uplo* = 'U', only the upper triangular part of *A* is used to define the elements of the symmetric matrix. If *uplo* = 'L', only the lower triangular part of *A* is used to define the elements of the Hermitian matrix.

ia,ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.If *desca*(*ctxt_*) is incorrect, p?heevx cannot guarantee correct error reporting

vl, vu (global)
REAL for pcheevx
DOUBLE PRECISION for pzheevx.
If *range* ='V', the lower and upper bounds of the interval to be searched for eigenvalues;
Not referenced if *range* ='A'or 'I'.

il, iu (global)
INTEGER. If *range* ='I', the indices of the smallest and largest eigenvalues to be returned.
Constraints:
$il \geq 1$
$\min(il,n) \leq iu \leq n$
Not referenced if *range* ='A'or 'V'.

abstol (global).
REAL for pcheevx
DOUBLE PRECISION for pzheevx.
If *jobz*='V', setting *abstol* to p?lamch(*context*, 'U') yields the most orthogonal eigenvectors.

The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*,*b*] of width less than or equal to
abstol + *eps* * max(|*a*|,|*b*|),
where *eps* is the machine precision. If *abstol* is less than or equal to zero, then *eps**norm(T) will be used in its place, where norm(T) is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*p?lamch('S') not zero.If this routine returns with ((mod(*info*,2).*ne*.0).*or*. (mod(*info*/8,2).*ne*.0)), indicating that some eigenvalues or eigenvectors did not converge, try setting *abstol* to 2*p?lamch('S').

orfac (global).
REAL for pcheevx
DOUBLE PRECISION for pzheevx.
Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within *tol*=*orfac**norm(*A*) of each other are to be reorthogonalized. However, if the workspace is insufficient (see *lwork*), *tol* may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of 10^3 is used if *orfac* is negative. *orfac* should be identical on all processes.

iz,jz (global) INTEGER. The row and column indices in the global array *z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *Z*.*descz*(*ctxt_*) must equal *desca*(*ctxt_*).

work (local)
COMPLEX for pcheevx
DOUBLE COMPLEX for pzheevx.
Array, DIMENSION (*lwork*).

lwork (local).
INTEGER. The dimension of the array *work*.
If only eigenvalues are requested:
lwork ≥ *n* + max(NB * (*np*0 + 1), 3)
If eigenvectors are requested:
lwork ≥ *n* + (*np*0+ *mq*0 + NB) * NB
with *nq*0 = numroc(*nn*, NB, 0, 0, NPCOL).

lwork >= 5**n* + max(5**nn*, *np*0 * *mq*0 + 2 * NB * NB) +iceil(*neig*, NPROW*NPCOL)**nn*

For optimal performance, greater workspace is needed, that is
lwork ≥ max(*lwork*, *nhetrd_lwork*)
where *lwork* is as defined above, and
nhetrd_lwork = *n* + 2*(*anb*+1)*(4**nps*+2) + (*nps* + 1) * *nps*

ictxt = *desca*(*ctxt_*)
anb = pjlaenv(*ictxt*, 3, 'pchettrd', 'L', 0, 0, 0, 0)
sqnpc = *sqrt*(*dble*(NPROW * NPCOL))
nps = max(numroc(*n*, 1, 0, 0, *sqnpc*), 2**anb*)

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.

rwork (local)
REAL for pcheevx
DOUBLE PRECISION for pzheevx.
Workspace array, DIMENSION (*lrwork*).

lrwork (local)
INTEGER. The dimension of the array *work*.
See below for definitions of variables used to define *lwork*.
If no eigenvectors are requested (*jobz* = 'N') then *lrwork* ≥ 5 * *nn* + 4* *n*
If eigenvectors are requested (*jobz* = 'V') then the amount of workspace required to guarantee that all eigenvectors are computed is:

lrwork ≥ 4**n* + max(5**nn*, *np*0 * *mq*0 + 2 * NB * NB) +iceil(*neig*, NPROW*NPCOL)**nn*

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following to *lrwork*: (*clustersize*-1)**n*

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:
{*w*(*k*),...,*w*(*k*+*clustersize*-1) |
w(*j*+1) ≤ *w*(*j*) + *orfac**2*norm(*A*)}

Variable definitions:
neig = number of eigenvectors requested
NB = *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*)
nn = max(*n*, NB, 2)
desca(*rsrc_*) = *desca*(*nb_*) = *descz*(*rsrc_*) = *descz*(*csrc_*) = 0
*np*0 = numroc(*nn*, NB, 0, 0, NPROW)
*mq*0 = numroc(max(*neig*, NB, 2), NB, 0, 0, NPCOL) iceil(*x*, *y*) is a ScaLAPACK function returning ceiling(*x*/*y*)

When *lrwork* is too small:
If *lwork* is too small to guarantee orthogonality, p?heevx attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.
If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*=-23 is returned. Note that when *range*='V', p?heevx does not know how many eigenvectors are requested until the eigenvalues are computed. Therefore, when *range*='V' and as long as *lwork* is large enough to allow p?heevx to compute the eigenvalues, p?heevx will compute the eigenvalues and as many eigenvectors as it can.

Relationship between workspace, orthogonality & performance:
If *clustersize* ≥ *n*/*sqrt*(NPROW*NPCOL), then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. In the limit (that is, *clustersize* = *n*-1) p?stein will perform no better than ?stein on 1 processor.
For *clustersize* = *n*/*sqrt*(NPROW*NPCOL) reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.
For *clustersize* > *n*/*sqrt*(NPROW*NPCOL) execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.

iwork (local) INTEGER. Workspace array.

liwork (local) INTEGER, dimension of *iwork*.
liwork ≥ 6 * *nnp*
Where: *nnp* = max(*n*, NPROW*NPCOL + 1, 4)
If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

m (global) INTEGER. The total number of eigenvalues found; $0 \le m \le n$.

nz (global) INTEGER.Total number of eigenvectors computed. $0 \le nz \le m$.
The number of columns of *z* that are filled.
If *jobz*.*ne*. 'V', *nz* is not referenced.
If *jobz*.*eq*. 'V', *nz* = *m* unless the user supplies insufficient space and p?heevx is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in *z* (*m*.*le*. *descz*(*n_*)) and sufficient workspace to compute them. (See *lwork*).p?heevx is always able to detect insufficient space without computation unless *range*.*eq*. 'V'.

w (global).
REAL for pcheevx
DOUBLE PRECISION for pzheevx
Array, DIMENSION (*n*).
The first *m* elements contain the selected eigenvalues in ascending order.

z (local).
COMPLEX for pcheevx
DOUBLE COMPLEX for pzheevx
Array, global dimension (*n*, *n*),
local dimension (*lld_z*, *LOCc*(*jz*+*n*-1))
If *jobz* ='V', then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
If *jobz* = 'N', then *z* is not referenced.

work(1)
On exit, returns workspace adequate workspace to allow optimal performance.

rwork
(local).

REAL for pcheevx
DOUBLE PRECISION for pzheevx
Array, DIMENSION (*lrwork*).
On return, *rwork(1)* contains the optimal amount of workspace required for efficient execution.
if *jobz*='N' *rwork(1)* = optimal amount of workspace required to compute eigenvalues efficiently.
if *jobz*='V' *rwork(1)* = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.
If *range*='V', it is assumed that all eigenvectors may be required.

iwork(1)
(local)
On return, *iwork(1)* contains the amount of integer workspace required.

ifail
(global) INTEGER.
Array, DIMENSION (*n*).
If *jobz* ='V', then on normal exit, the first *m* elements of *ifail* are zero. If (mod(*info*,2).*ne*.0) on exit, then *ifail* contains the indices of the eigenvectors that failed to converge.
If *jobz* = 'N', then *ifail* is not referenced.

iclustr
(global) INTEGER.
Array, DIMENSION (2*NPROW*NPCOL)
This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*).Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2**i*-1) to *iclustr*(2**i*), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr*() is a zero terminated array. (*iclustr*(2**k*).*ne*.0.*and*. *iclustr*(2**k*+1).*eq*.0) if and only if *k* is the number of clusters.
iclustr is not referenced if *jobz* = 'N'

gap
(global)
REAL for pcheevx
DOUBLE PRECISION for pzheevx
Array, DIMENSION (NPROW*NPCOL)
This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters

indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the i^{th} cluster may be as high as (*C* * *n*) / *gap*(*i*) where *C* is a small constant.

info (global) INTEGER.
If *info* = 0, the execution is successful.
If *info*< 0:
If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
If *info* > 0:
if (mod(*info*,2).*ne*.0), then one or more eigenvectors failed to converge. Their indices are stored in *ifail*. Ensure *abstol*=2.0*p?lamch('U')
if (mod(*info*/2,2).*ne*.0),then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace.The indices of the clusters are stored in the array *iclustr*.
if (mod(*info*/4,2).*ne*.0), then space limit prevented p?syevx from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.
if (mod(*info*/8,2).*ne*.0), then p?stebz failed to compute eigenvalues. Ensure *abstol*=2.0*p?lamch('U').

p?gesvd

Computes the singular value decomposition of a general matrix, optionally computing the left and/or right singular vectors.

Syntax

```
call psgesvd ( jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu,
    vt, ivt, jvt, descvt, work, lwork, info)
call pdgesvd ( jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu,
    vt, ivt, jvt, descvt, work, lwork, info)
```

Description

This routine computes the singular value decomposition (SVD) of an *m*-by-*n* matrix *A*, optionally computing the left and/or right singular vectors. The SVD is written

$$A = U \Sigma V^T$$

where Σ 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 Σ are the singular values of *A*and the columns of *U* and *V* are the corresponding right and left singular vectors, respectively. The singular values are returned in array *s* in decreasing order and only the first min(*m*,*n*) columns of *U* and rows of *vt* = V^T are computed.

Input Parameters

mp = number of local rows in *A* and *U*
nq = number of local columns in *A* and *VT*
size = min(*m*,*n*)
sizeq = number of local columns in *U*
sizep = number of local rows in *VT*

jobu (global).CHARACTER*1.
Specifies options for computing all or part of the matrix *U*.

If *jobu* ='V', the first *size* columns of *U* (the left singular vectors) are returned in the array *u*;
if *jobu* ='N', no columns of *U* (no left singular vectors) are computed.

jobvt (global) CHARACTER*1.
Specifies options for computing all or part of the matrix V^T.

If *jobvt* ='V', the first *size* rows of V^T (the right singular vectors) are returned in the array *vt*;
if *jobvt* ='N', no rows of V^T(no right singular vectors) are computed.

m (global) INTEGER. The number of rows of the matrix *A* ($m \geq 0$).

n (global) INTEGER. The number of columns in *A* ($n \geq 0$).

a (local).
DOUBLE PRECISION for psgesvd and pdgesvd
Block cyclic array, global dimension (*m*, *n*), local dimension (*mp*, *nq*).

work(*lwork*) is a workspace array.

ia,ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

iu,ju — (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *U*, respectively.

descu — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *U*.

ivt,jvt — (global) INTEGER. The row and column indices in the global array *vt* indicating the first row and the first column of the submatrix *VT*, respectively.

descvt — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *VT*.

work — (local)
DOUBLE PRECISION for psgesvd and pdgesvd
Workspace array, dimension (*lwork*)

lwork — (local)
INTEGER. The dimension of the array *work*;

lwork > 2 + 6**sizeb* + max(*watobd*, *wbdtosvd*),

where *sizeb* = max(*m*,*n*), and *watobd* and *wbdtosvd* refer, respectively, to the workspace required to bidiagonalize the matrix A and to go from the bidiagonal matrix to the singular value decomposition *U S VT*.

For *watobd*, the following holds:

watobd = max(max(*wpslange*,*wpsgebrd*),
max(*wpslared2d*,*wpslared1d*)),

where *wpslange*, *wpslared1d*, *wpslared2d*, *wpsgebrd* are the workspaces required respectively for the subprograms pslange, pslared1d, pslared2d, psgebrd. Using the standard notation

mp = numroc(*m*, *mb*, MYROW, *desca*(*ctxt_*), NPROW),
nq = numroc(*n*, NB, MYCOL, *desca*(*lld_*), NPCOL),

the workspaces required for the above subprograms are

wpslange = *mp*,
wpslared1d = *nq*0,
wpslared2d = *mp*0,
wpsgebrd = NB*(*mp* + *nq* + 1) + *nq*,

where *nq*0 and *mp*0 refer, respectively, to the values obtained at MYCOL = 0 and MYROW = 0. In general, the upper limit for the workspace is given by a workspace required on processor (0,0):
watobd ≤ NB*(*mp*0 + *nq*0 + 1) + *nq*0.

In case of a homogeneous process grid this upper limit can be used as an estimate of the minimum workspace for every processor.

For *wbdtosvd*, the following holds:

wbdtosvd = *size**(*wantu** *nru* + *wantvt**ncvt) + max(*wsbdsqr*, max(*wantu**wpsormbrqln*, *wantvt**wpsormbrprt*)),

where

1, if left(right) singular vectors are wanted *wantu*(*wantvt*) = 0,otherwise and *wsbdsqr*, *wpsormbrqln* and *wpsormbrprt* refer respectively to the workspace required for the subprograms sbdsqr, p?ormbr(*qln*), and p?ormbr(*prt*), where *qln* and *prt* are the values of the arguments *vect*, *side*, and *trans* in the call to p?ormbr. *nru* is equal to the local number of rows of the matrix *U* when distributed 1-dimensional "column" of processes. Analogously, *ncvt* is equal to the local number of columns of the matrix *VT* when distributed across 1-dimensional "row" of processes. Calling the LAPACK procedure sbdsqr requires

wsbdsqr = max(1, 2**size* + (2**size* - 4)* max(*wantu*, *wantvt*))

on every processor. Finally,

wpsormbrqln = max((NB*(NB-1))/2,
(*sizeq*+*mp*)*NB)+NB*NB,
wpsormbrprt = max((*mb**(*mb*-1))/2,
(*sizep*+*nq*)**mb*)+*mb***mb*,

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum size for the work array. The required workspace is returned as the first element of *work* and no error message is issued by pxerbla.

Output Parameters

a — On exit, the contents of *a* are destroyed.

s — (global).
DOUBLE PRECISION for psgesvd and pdgesvd.
Array, DIMENSION (*size*).
Contains the singular values of *A* sorted so that
s(i) ≥ *s*(i+1).

u (local).
DOUBLE PRECISION for psgesvd and pdgesvd
local dimension (*mp*, *sizeq*), global dimension (*m*, *size*)
if *jobu* = 'V', *u* contains the first min(*m*,*n*) columns of *U* If *jobu* = 'N' or 'O', *u* is not referenced.

vt (local)
DOUBLE PRECISION for psgesvd and pdgesvd
local dimension (*sizep*, *nq)*, global dimension (*size*, *n*)
if *jobvt* = 'V',*VT* contains the first *size* rows of V^T
If *jobu* = 'N',*VT* is not referenced.

work On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

rwork On exit (for complex flavors), if *info* > 0, *rwork*(1:min(*m*,*n*)-1) contains the unconverged superdiagonal elements of an upper bidiagonal matrix *B* whose diagonal is in *s* (not necessarily sorted). *B* satisfies *A* = *u* * *B* * *vt*, so it has the same singular values as *A*, and singular vectors related by *u* and *vt*.

info (global) INTEGER.
If *info* = 0, the execution is successful.
If *info* < 0, If *info*= *-i*, the *i*th parameter had an illegal value.
If *info* >0 *i*, then if p?bdsqr did not converge,
If *info* = min(*m*,*n*) + 1, then p?gesvd has detected heterogeneity by finding that eigenvalues were not identical across the process grid. In this case, the accuracy of the results from p?gesvd cannot be guaranteed.

p?sygvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

```
call pssygvx(ibtype, jobz, range, uplo, n, a, ia, ja, desca, b, ib, jb,
    descb, vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz,
    work, lwork, iwork, liwork, ifail, iclustr, gap, info)
call pdsygvx(ibtype, jobz, range, uplo, n, a, ia, ja, desca, b, ib, jb,
    descb, vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz,
    work, lwork, iwork, liwork, ifail, iclustr, gap, info)
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$\mathrm{sub}(A)x = \lambda\, \mathrm{sub}(B)x,\quad \mathrm{sub}(A)\,\mathrm{sub}(B)x = \lambda x,\ \text{ or } \mathrm{sub}(B)\,\mathrm{sub}(A)x = \lambda x.$$

Here sub(A) denoting A(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) is assumed to symmetric and sub(B) denoting B(*ib*:*ib*+*n*-1, *jb*:*jb*+*n*-1) is also positive definite.

Input Parameters

ibtype (global) `INTEGER`. Must be 1 or 2 or 3.
Specifies the problem type to be solved:
if *ibtype* = 1, the problem type is
$\mathrm{sub}(A)x = \lambda\,\mathrm{sub}(B)x$;
if *ibtype* = 2, the problem type is
$\mathrm{sub}(A)\mathrm{sub}(B)x = \lambda x$;
if *ibtype* = 3, the problem type is
$\mathrm{sub}(B)\,\mathrm{sub}(A)x = \lambda x$.

jobz (global).`CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* =`'N'`, then compute eigenvalues only.
If *jobz* =`'V'`, then compute eigenvalues and eigenvectors.

range (global).
`CHARACTER*1`. Must be `'A'` or `'V'` or `'I'`.

If *range* ='A', the routine computes all eigenvalues.
If *range* ='V', the routine computes eigenvalues in the interval: [*vl*, *vu*]
If *range* ='I', the routine computes eigenvalues with indices *il* through *iu*.

uplo (global).
CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', arrays *a* and *b* store the upper triangles of sub(*A*) and sub (*B*);
If *uplo* = 'L', arrays *a* and *b* store the lower triangles of sub(*A*) and sub (*B*).

n (global).
INTEGER. The order of the matrices sub(*A*) and sub (*B*) $n \geq 0$.

a (local)
REAL for pssygvx
DOUBLE PRECISION for pdsygvx.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, this array contains the local pieces of the *n*-by-*n* symmetric distributed matrix sub(*A*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix.If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix.

ia,ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.If *desca*(*ctxt_*) is incorrect, p?sygvx cannot guarantee correct error reporting.

b (local).
REAL for pssygvx
DOUBLE PRECISION for pdsygvx.
Pointer into the local memory to an array of dimension (*lld_b*, *LOCc*(*jb*+*n*-1)). On entry, this array contains the local pieces of the *n*-by-*n* symmetric distributed matrix sub(*B*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*B*) contains the upper triangular part of the matrix.If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix.

ib,jb (global) INTEGER. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*. *descb*(*ctxt_*) must be equal to *desca*(*ctxt_*).

vl, vu (global)
REAL for pssygvx
DOUBLE PRECISION for pdsygvx.
If *range* ='V', the lower and upper bounds of the interval to be searched for eigenvalues.

If *range* ='A' or 'I', *vl* and *vu* are not referenced.

il, iu (global)
INTEGER.
If *range* ='I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: *il* ≥ 1, min (*il*, *n*) ≤ *iu* ≤ *n*

If *range* ='A' or 'V', *il* and *iu* are not referenced.

abstol (global)
REAL for pssygvx
DOUBLE PRECISION for pdsygvx.
If *jobz*='V', setting *abstol* to p?lamch(*context*, 'U') yields the most orthogonal eigenvectors.
The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*,*b*] of width less than or equal to

abstol + *eps* * max(|*a*|,|*b*|),

where *eps* is the machine precision. If *abstol* is less than or equal to zero, then *eps**norm(T) will be used in its place, where norm(T) is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*p?lamch('S') not zero. If this routine returns with ((mod(*info*,2).*ne*.0).*or*. * (*mod*(*info*/8,2).*ne*.0)), indicating that some eigenvalues or eigenvectors did not converge, try setting *abstol* to 2*p?lamch('S').

orfac (global).
REAL for pssygvx
DOUBLE PRECISION for pdsygvx.
Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within *tol*=*orfac**norm(*A*) of each other are to be reorthogonalized. However, if the workspace is insufficient (see *lwork*), *tol* may be decreased until all eigenvectors to be reorthogonalized

can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of 10^{-3} is used if *orfac* is negative. *orfac* should be identical on all processes.

iz,jz (global) INTEGER. The row and column indices in the global array *z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *Z*.*descz*(*ctxt_*) must equal *desca*(*ctxt_*).

work (local)
REAL for pssygvx
DOUBLE PRECISION for pdsygvx.
Workspace array, dimension of the (*lwork*)

lwork (local)
INTEGER.
See below for definitions of variables used to define *lwork*.
If no eigenvectors are requested (*jobz* = 'N') then *lwork* ≥ 5 * *n* + max(5 * *nn*, NB * (*np*0 + 1)).
If eigenvectors are requested (*jobz* = 'V') then the amount of workspace required to guarantee that all eigenvectors are computed is:

lwork >= 5**n* + max(5**nn*, *np*0 * *mq*0 + 2 * NB * NB) +iceil(*neig*, NPROW*NPCOL)**nn*

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following to *lwork*:
(*clustersize*-1)**n*
where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:
{*w*(*k*),...,*w*(*k*+*clustersize*-1) |
w(*j*+1) ≤ *w*(*j*) + *orfac**2*norm(*A*)}

Variable definitions:
neig = number of eigenvectors requested
NB = *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*)
nn = max(*n*, NB, 2)
desca(*rsrc_*) = *desca*(*nb_*) = *descz*(*rsrc_*) = *descz*(*csrc_*) = 0
*np*0 = numroc(*nn*, NB, 0, 0, NPROW)
*mq*0 = numroc(max(*neig*, NB, 2), NB, 0, 0, NPCOL) iceil(*x*, *y*) is a ScaLAPACK function returning ceiling(*x*/*y*)

When *lwork* is too small:
If *lwork* is too small to guarantee orthogonality, p?syevx attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.
If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*=-23 is returned. Note that when *range*='V', p?sygvx does not know how many eigenvectors are requested until the eigenvalues are computed. Therefore, when *range*='V'and as long as *lwork* is large enough to allow p?sygvx to compute the eigenvalues, p?sygvx will compute the eigenvalues and as many eigenvectors as it can.

Relationship between workspace, orthogonality & performance:
Greater performance can be achieved if adequate workspace is provided. On the other hand, in some situations, performance can decrease as the workspace provided increases above the workspace amount shown below:

For optimal performance, greater workspace may be needed, that is,
lwork ≥ max(*lwork*, 5**n* + *nsytrd_lwopt*, *nsygst_lwopt*)
Where:
lwork, as defined previously, depends upon the number of eigenvectors requested, and
nsytrd_lwopt = *n* + 2*(*anb*+1)*(4**nps*+2) + (*nps* + 3) * *nps*

nsygst_lwopt = 2**np*0*NB + *nq*0*NB + NB*NB
anb = pjlaenv(*desca*(*ctxt_*), 3, p?syttrd', 'L', 0, 0, 0, 0)
sqnpc = int(*sqrt*(*dble*(NPROW * NPCOL)))
nps = max(numroc(*n*, 1, 0, 0, *sqnpc*), 2**anb*)
NB = *desca*(*mb_*)
*np*0 =numroc(*n*, NB, 0, 0, NPROW)
*nq*0 = numroc(*n*, NB, 0, 0, NPCOL)

numroc is a ScaLAPACK tool functions;
pjlaenv is a ScaLAPACK environmental inquiry function
MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

For large *n*, no extra workspace is needed, however the biggest boost in performance comes for small *n*, so it is wise to provide the extra workspace (typically less than a Megabyte per process).

If *clustersize* ≥ *n*/*sqrt*(NPROW*NPCOL), then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. In the limit (that is, *clustersize* = *n*-1) p?stein will

perform no better than `?stein` on 1 processor.
For *clustersize* = *n*/*sqrt*(NPROW*NPCOL) reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.
For *clustersize* > *n*/*sqrt*(NPROW*NPCOL) execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by `pxerbla`.

iwork (local) `INTEGER`. Workspace array.

liwork (local) `INTEGER`, dimension of *iwork*.
liwork ≥ 6 * *nnp*
Where:
nnp = max(*n*, NPROW*NPCOL + 1, 4)
If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a On exit, if *jobz* = 'V', then if *info* = 0, sub(*A*) contains the distributed matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:
if *ibtype* = 1 or 2,
Z^T*sub(*B*)*Z = *i*;
if *ibtype* = 3, Z^T*inv(sub(*B*))*Z = *i*.
If *jobz* = 'N', then on exit the upper triangle (if *uplo*='U') or the lower triangle (if *uplo*='L') of sub(*A*), including the diagonal, is destroyed.

b On exit, if *info* ≤ *n*, the part of sub(*B*) containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization sub(*B*) = U^TU or sub(*B*) = $L L^T$.

m (global)
`INTEGER`. The total number of eigenvalues found,
$0 \le m \le n$.

nz (global)
INTEGER.
Total number of eigenvectors computed. $0 \le nz \le m$. The number of columns of *z* that are filled.
If *jobz*.*ne*. 'V', *nz* is not referenced.
If *jobz*.*eq*. 'V', *nz* = *m* unless the user supplies insufficient space and p?sygvx is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in *z* (*m*.*le*. *descz*(*n_*)) and sufficient workspace to compute them. (See *lwork* below.) p?sygvx is always able to detect insufficient space without computation unless
range.*eq*. 'V'.

w (global)
REAL for pssygvx
DOUBLE PRECISION for pdsygvx.
Array, DIMENSION (*n*).
On normal exit, the first *m* entries contain the selected eigenvalues in ascending order.

z (local).
REAL for pssygvx
DOUBLE PRECISION for pdsygvx.
global dimension (*n*, *n*), local dimension (*lld_z*, *LOCc*(*jz*+*n*-1)).
If *jobz* = 'V', then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
If *jobz* = 'N', then *z* is not referenced.

work if*jobz*='N' *work(1)* = optimal amount of workspace required to compute eigenvalues efficiently
if *jobz* = 'V' *work(1)* = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.
If *range*='V', it is assumed that all eigenvectors may be required.

ifail (global)
INTEGER.
Array, DIMENSION (*n*).
ifail provides additional information when *info*.*ne*. 0

If (*mod*(*info*/16,2).*ne*.0) then *ifail(1)* indicates the order of the smallest minor which is not positive definite. If (*mod*(*info*,2).*ne*.0) on exit, then *ifail* contains the indices of the eigenvectors that failed to converge.

If neither of the above error conditions hold and *jobz* = 'V', then the first *m* elements of *ifail* are set to zero.

iclustr (global)
INTEGER.
Array, DIMENSION (2*NPROW*NPCOL).This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2*i-1) to *iclustr*(2*i), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr*() is a zero terminated array. (*iclustr*(2*k).*ne*.0.*and*. *iclustr*(2*k+1).*eq*.0) if and only if k is the number of clusters *iclustr* is not referenced if *jobz* = 'N'.

gap (global)
REAL for pssygvx
DOUBLE PRECISION for pdsygvx.
Array, DIMENSION (NPROW*NPCOL).
This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the i^{th} cluster may be as high as (C * n) / *gap*(i) where C is a small constant.

info (global)
INTEGER.
If *info* = 0, the execution is successful.
If *info* <0: the *i*th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
If *info* > 0:
if (mod(*info*,2).*ne*.0), then one or more eigenvectors failed to converge. Their indices are stored in *ifail*.
if (mod(*info*,2,2).*ne*.0),then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.
if (mod(*info*/4,2).*ne*.0), then space limit prevented p?sygvx from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors

computed is returned in *nz*.
if (mod(*info*/8,2).*ne*.0), then p?stebz failed to compute eigenvalues.
if (mod(*info*/16,2).*ne*.0), then *B* was not positive definite. *ifail(1)* indicates the order of the smallest minor which is not positive definite.

p?hegvx

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem.

Syntax

```
call pchegvx ( ibtype, jobz, range, uplo, n, a, ia, ja, desca, b, ib, jb,
    descb, vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz,
    work, lwork, rwork, lrwork, iwork, liwork, ifail, iclustr, gap, info)
call pzhegvx ( ibtype, jobz, range, uplo, n, a, ia, ja, desca, b, ib, jb,
    descb, vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz,
    work, lwork, rwork, lrwork, iwork, liwork, ifail, iclustr, gap, info)
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$$\text{sub}(A)x = \lambda\,\text{sub}(B)x, \quad \text{sub}(A)\text{sub}(B)x = \lambda x, \quad \text{or sub}(B)\text{sub}(A)x = \lambda x.$$

Here sub (A) denoting A(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) and sub(B) are assumed to be Hermitian and sub(B) denoting B(*ib*:*ib*+*n*-1, *jb*:*jb*+*n*-1) is also positive definite.

Input Parameters

ibtype (global) INTEGER. Must be 1 or 2 or 3.
Specifies the problem type to be solved:
if *ibtype* = 1, the problem type is
sub(A)$x = \lambda$ sub(B)x;
if *ibtype* = 2, the problem type is
sub(A)sub(B)$x = \lambda x$;
if *ibtype* = 3, the problem type is
sub(B) sub(A)$x = \lambda x$.

jobz	(global).CHARACTER*1. Must be 'N' or 'V'. If *jobz* ='N', then compute eigenvalues only. If *jobz* ='V', then compute eigenvalues and eigenvectors.
range	(global). CHARACTER*1. Must be 'A' or 'V' or 'I'. If *range* ='A', the routine computes all eigenvalues. If *range* ='V', the routine computes eigenvalues in the interval: [*vl*, *vu*] If *range* ='I', the routine computes eigenvalues with indices *il* through *iu*.
uplo	(global). CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', arrays *a* and *b* store the upper triangles of sub(*A*) and sub (*B*); If *uplo* = 'L', arrays *a* and *b* store the lower triangles of sub(*A*) and sub (*B*).
n	(global). INTEGER. The order of the matrices sub(*A*) and sub (*B*) ($n \geq 0$).
a	(local) COMPLEX for pchegvx DOUBLE COMPLEX for pzhegvx. Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, this array contains the local pieces of the *n*-by-*n* Hermitian distributed matrix sub(*A*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix.If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix.
ia,ja	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.If *desca*(*ctxt_*) is incorrect, p?hegvx cannot guarantee correct error reporting.
b	(local). COMPLEX for pchegvx DOUBLE COMPLEX for pzhegvx. Pointer into the local memory to an array of dimension (*lld_b*, *LOCc*(*jb*+*n*-1)). On entry, this array contains the local pieces of the *n*-by-*n* Hermitian distributed matrix sub(*B*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*B*) contains the upper triangular part of the matrix.If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*B*) contains the lower triangular part of the matrix.

ib,jb — (global) INTEGER. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.

descb — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix B. *descb*(*ctxt_*) must be equal to *desca*(*ctxt_*).

vl, vu — (global)
REAL for pchegvx
DOUBLE PRECISION for pzhegvx.
If *range* ='V', the lower and upper bounds of the interval to be searched for eigenvalues.

If *range* ='A' or 'I', *vl* and *vu* are not referenced.

il, iu — (global)
INTEGER.
If *range* ='I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: $il \geq 1$, $\min(il, n) \leq iu \leq n$

If *range* ='A' or 'V', *il* and *iu* are not referenced.

abstol — (global)
REAL for pchegvx
DOUBLE PRECISION for pzhegvx.
If *jobz*='V', setting *abstol* to p?lamch(*context*, 'U') yields the most orthogonal eigenvectors.
The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*,*b*] of width less than or equal to

abstol + *eps* * max(|*a*|,|*b*|),

where *eps* is the machine precision. If *abstol* is less than or equal to zero, then *eps**norm(T) will be used in its place, where norm(T) is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*p?lamch('S') not zero. If this routine returns with ((mod(*info*,2).*ne*.0).*or*. * (*mod*(*info*/8,2).*ne*.0)), indicating that some eigenvalues or eigenvectors did not converge, try setting *abstol* to 2*p?lamch('S').

orfac (global).
REAL for pchegvx
DOUBLE PRECISION for pzhegvx.
Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within *tol*=*orfac**norm(*A*) of each other are to be reorthogonalized. However, if the workspace is insufficient (see *lwork*), *tol* may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of 10^{-3} is used if *orfac* is negative. *orfac* should be identical on all processes.

iz,jz (global) INTEGER. The row and column indices in the global array *z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *Z*.*descz*(*ctxt_*) must equal *desca*(*ctxt_*).

work (local)
COMPLEX for pchegvx
DOUBLE COMPLEX for pzhegvx.
Workspace array, dimension (*lwork*)

lwork (local).
INTEGER. The dimension of the array *work*.
If only eigenvalues are requested:
lwork ≥ *n* + max(NB * (*np*0 + 1), 3)
If eigenvectors are requested:
lwork ≥ *n* + (*np*0+ *mq*0 + NB) * NB
with *nq*0 = numroc(*nn*, NB, 0, 0, NPCOL).

For optimal performance, greater workspace is needed, that is
lwork ≥ max(*lwork*, *n*, *nhetrd_lwopt, nhegst_lwopt)*
where *lwork* is as defined above, and
nhetrd_lwork = 2*(*anb*+1)*(4**nps*+2) + (*nps* + 1) * *nps*
nhegst_lwopt = 2**np*0*NB + *nq*0*NB + NB*NB

NB = *desca*(*mb_*)
*np*0 = numroc(*n*, NB, 0, 0, NPROW)
*nq*0 = numroc(*n*, NB, 0, 0, NPCOL)
ictxt = *desca*(*ctxt_*)
anb = pjlaenv(*ictxt*, 3, 'p?hettrd', 'L', 0, 0, 0, 0)
sqnpc = *sqrt*(*dble*(NPROW * NPCOL))
nps = max(numroc(*n*, 1, 0, 0, *sqnpc*), 2**anb*)

numroc is a ScaLAPACK tool functions;
pjlaenv is a ScaLAPACK environmental inquiry function MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.

rwork (local)
REAL for pchegvx
DOUBLE PRECISION for pzhegvx.
Workspace array, DIMENSION (*lrwork*).

lrwork (local)
INTEGER. The dimension of the array *rwork*.
See below for definitions of variables used to define *lrwork*.
If no eigenvectors are requested (*jobz* = 'N') then *lrwork* ≥ 5 * *nn* + 4* *n*
If eigenvectors are requested (*jobz* = 'V') then the amount of workspace required to guarantee that all eigenvectors are computed is:

lrwork ≥ 4**n* + max(5**nn*, *np*0 * *mq*0) +
iceil(*neig*, NPROW*NPCOL)**nn*

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following to *lrwork*:
(*clustersize*-1)**n*
where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:
{*w*(*k*),...,*w*(*k*+*clustersize*-1) |
w(*j*+1) ≤ *w*(*j*) + *orfac**2*norm(*A*)}

Variable definitions:
neig = number of eigenvectors requested
NB = *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*)
nn = max(*n*, NB, 2)
desca(*rsrc_*) = *desca*(*nb_*) = *descz*(*rsrc_*) = *descz*(*csrc_*) = 0
*np*0 = numroc(*nn*, NB, 0, 0, NPROW)
*mq*0 = numroc(max(*neig*, NB, 2), NB, 0, 0, NPCOL) iceil(*x*, *y*) is a ScaLAPACK function returning ceiling(*x*/*y*)

When *lrwork* is too small:
If *lwork* is too small to guarantee orthogonality, `p?hegvx` attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.
If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*=-25 is returned. Note that when *range*='V', `p?hegvx` does not know how many eigenvectors are requested until the eigenvalues are computed. Therefore, when *range*='V'and as long as *lwork* is large enough to allow `p?hegvx` to compute the eigenvalues, `p?hegvx` will compute the eigenvalues and as many eigenvectors as it can.

Relationship between workspace, orthogonality & performance:
If *clustersize* ≥ *n*/*sqrt*(`NPROW*NPCOL`), then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. In the limit (that is, *clustersize* = *n*-1) `p?stein` will perform no better than `?stein` on 1 processor.
For *clustersize* = *n*/*sqrt*(`NPROW*NPCOL`) reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.
For *clustersize* > *n*/*sqrt*(`NPROW*NPCOL`) execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If *lwork* = -1, then *lrwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by `pxerbla`.

iwork (local) `INTEGER`. Workspace array.

liwork (local) `INTEGER`, dimension of *iwork*.
liwork ≥ 6 * *nnp*
Where: *nnp* = `max`(*n*, `NPROW*NPCOL` + 1, 4)
If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a On exit, if *jobz* = 'V', then if *info* = 0, sub(*A*) contains the distributed matrix *Z* of eigenvectors.
The eigenvectors are normalized as follows:

if *ibtype* = 1 or 2, Z^H*sub(*B*)*Z = *i*;
if *ibtype* = 3, Z^H*inv(sub(*B*))*Z = *i*.
If *jobz* = 'N', then on exit the upper triangle (if *uplo*='U') or the lower triangle (if *uplo*='L') of sub(*A*), including the diagonal, is destroyed.

b — On exit, if *info* ≤ *n*, the part of sub(*B*) containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization sub(*B*) = U^HU or sub(*B*) = $L L^H$.

m — (global)
INTEGER. The total number of eigenvalues found,
0 ≤ *m* ≤ *n*.

nz — (global)
INTEGER.
Total number of eigenvectors computed. 0 ≤ *nz* ≤ *m*. The number of columns of *z* that are filled.
If *jobz.ne*. 'V', *nz* is not referenced.
If *jobz.eq*. 'V', *nz* = *m* unless the user supplies insufficient space and p?hegvx is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in *z* (*m.le*. *descz*(*n_*)) and sufficient workspace to compute them. (See *lwork* below.) p?hegvx is always able to detect insufficient space without computation unless
range.eq. 'V'.

w — (global)
REAL for pchegvx
DOUBLE PRECISION for pzhegvx.
Array, DIMENSION (*n*).
On normal exit, the first *m* entries contain the selected eigenvalues in ascending order.

z — (local).
COMPLEX for pchegvx
DOUBLE COMPLEX for pzhegvx.
global dimension (*n*, *n*), local dimension (*lld_z*, *LOCc*(*jz*+*n*-1)).
If *jobz* = 'V', then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
If *jobz* = 'N', then *z* is not referenced.

work On exit, *work(1)* returns the optimal amount of workspace.

rwork On exit, *rwork(1)* contains the amount of workspace required for optimal efficiency
if *jobz*='N' *rwork(1)* = optimal amount of workspace required to compute eigenvalues efficiently
if *jobz*='V' *rwork(1)* = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.
If *range*='V', it is assumed that all eigenvectors may be required when computing optimal workspace.

ifail (global)
INTEGER.
Array, DIMENSION (*n*).
ifail provides additional information when
info.*ne*. 0
If (*mod*(*info*/16,2).*ne*.0) then *ifail(1)* indicates the order of the smallest minor which is not positive definite. If (*mod*(*info*,2).*ne*.0) on exit, then *ifail(1)* contains the indices of the eigenvectors that failed to converge.

If neither of the above error conditions hold and
jobz = 'V', then the first *m* elements of *ifail* are set to zero.

iclustr (global)
INTEGER.
Array, DIMENSION (2*NPROW*NPCOL).This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2**i*-1) to *iclustr*(2**i*), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr*() is a zero terminated array.
(*iclustr*(2**k*).*ne*.0.*and*. *iclustr*(2**k*+1).*eq*.0) if and only if *k* is the number of clusters *iclustr* is not referenced if *jobz* = 'N'.

gap (global)
REAL for pchegvx
DOUBLE PRECISION for pzhegvx.
Array, DIMENSION (NPROW*NPCOL).
This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters

indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the i^{th} cluster may be as high as (*C* * *n*) / *gap*(*i*) where *C* is a small constant.

info (global)
INTEGER.
If *info* = 0, the execution is successful.
If *info* <0: the *i*th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
If *info* > 0:
if (mod(*info*,2).*ne*.0), then one or more eigenvectors failed to converge. Their indices are stored in *ifail*.
if (mod(*info*,2,2).*ne*.0), then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.
if (mod(*info*/4,2).*ne*.0), then space limit prevented p?sygvx from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.
if (mod(*info*/8,2).*ne*.0), then p?stebz failed to compute eigenvalues.
if (mod(*info*/16,2).*ne*.0), then *B* was not positive definite. *ifail(1)* indicates the order of the smallest minor which is not positive definite.

ScaLAPACK Auxiliary and Utility Routines 7

This chapter describes the Intel® Math Kernel Library implementation of ScaLAPACK Auxiliary Routines and Utility Functions and Routines. The library includes routines for both real and complex data.

NOTE. ScaLAPACK routines are provided with Intel® Cluster MKL product only which is a superset of Intel MKL.

Routine naming conventions, mathematical notation, and matrix storage schemes used for ScaLAPACK auxiliary and utility routines are the same as described in previous chapters. Some routines and functions may have combined character codes, such as `sc` or `dz`. For example, the routine `pscsum1` uses a complex input array and returns a real value.

Auxiliary Routines

Table 7-1 ScaLAPACK Auxiliary Routines

Routine Name	Data Types	Description
`p?lacgv`	`c,z`	Conjugates a complex vector.
`p?max1`	`c,z`	Finds the index of the element whose real part has maximum absolute value (similar to the Level 1 PBLAS `p?amax`, but using the absolute value to the real part).
`?combamax1`	`c,z`	Finds the element with maximum real part absolute value and its corresponding global index.

Table 7-1 ScaLAPACK Auxiliary Routines (continued)

Routine Name	Data Types	Description
p?sum1	sc,dz	Forms the 1-norm of a complex vector similar to Level 1 PBLAS p?asum, but using the true absolute value.
p?dbtrsv	s,d,c,z	Computes an *LU* factorization of a general tridiagonal matrix with no pivoting. The routine is called by p?dbtrs.
p?dttrsv	s,d,c,z	Computes an *LU* factorization of a general band matrix, using partial pivoting with row interchanges. The routine is called by p?dttrs.
p?gebd2	s,d,c,z	Reduces a general rectangular matrix to real bidiagonal form by an orthogonal/unitary transformation (unblocked algorithm).
p?gehd2	s,d,c,z	Reduces a general matrix to upper Hessenberg form by an orthogonal/unitary similarity transformation (unblocked algorithm).
p?gelq2	s,d,c,z	Computes an *LQ* factorization of a general rectangular matrix (unblocked algorithm).
p?geql2	s,d,c,z	Computes a *QL* factorization of a general rectangular matrix (unblocked algorithm).
p?geqr2	s,d,c,z	Computes a *QR* factorization of a general rectangular matrix (unblocked algorithm).
p?gerq2	s,d,c,z	Computes an *RQ* factorization of a general rectangular matrix (unblocked algorithm).
p?getf2	s,d,c,z	Computes an *LU* factorization of a general matrix, using partial pivoting with row interchanges (local blocked algorithm).
p?labrd	s,d,c,z	Reduces the first *nb* rows and columns of a general rectangular matrix A to real bidiagonal form by an orthogonal|unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A.
p?lacon	s,d,c,z	Estimates the 1-norm of a square matrix, using the reverse communication for evaluating matrix-vector products.
p?laconsb	s,d	Looks for two consecutive small subdiagonal elements.
p?lacp2	s,d,c,z	Copies all or part of a distributed matrix to another distributed matrix.
p?lacp3	s,d	Copies from a global parallel array into a local replicated array or vice versa.
p?lacpy	s,d,c,z	Copies all or part of one two-dimensional array to another.
p?laevswp	s,d,c,z	Moves the eigenvectors from where they are computed to ScaLAPACK standard block cyclic array.

Table 7-1 ScaLAPACK Auxiliary Routines (continued)

Routine Name	Data Types	Description
p?lahrd	s,d,c,z	Reduces the first *nb* columns of a general rectangular matrix A so that elements below the k^{th} subdiagonal are zero, by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A.
p?laiect	s,d,c,z	Exploits IEEE arithmetic to accelerate the computations of eigenvalues. (C interface function).
p?lange	s,d,c,z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a general rectangular matrix.
p?lanhs	s,d,c,z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of an upper Hessenberg matrix.
p?lansy, p?lanhe	s,d,c,z /c,z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a real symmetric or complex Hermitian matrix.
p?lantr	s,d,c,z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a triangular matrix.
p?lapiv	s,d,c,z	Applies a permutation matrix to a general distributed matrix, resulting in row or column pivoting.
p?laqge	s,d,c,z	Scales a general rectangular matrix, using row and column scaling factors computed by p?geequ.
p?laqsy	s,d,c,z	Scales a symmetric/Hermitian matrix, using scaling factors computed by p?poequ.
p?lared1d	s,d	Redistributes an array assuming that the input array *bycol* is distributed across rows and that all process columns contain the same copy of *bycol*.
p?lared2d	s,d	Redistributes an array assuming that the input array *byrow* is distributed across columns and that all process rows contain the same copy of *byrow* .
p?larf	s,d,c,z	Applies an elementary reflector to a general rectangular matrix.
p?larfb	s,d,c,z	Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.
p?larfc	c,z	Applies the conjugate transpose of an elementary reflector to a general matrix.

Table 7-1 ScaLAPACK Auxiliary Routines (continued)

Routine Name	Data Types	Description
p?larfg	s,d,c,z	Generates an elementary reflector (Householder matrix).
p?larft	s,d,c,z	Forms the triangular vector T of a block reflector $H=I-VTV^H$.
p?larz	s,d,c,z	Applies an elementary reflector as returned by `p?tzrzf` to a general matrix.
p?larzb	s,d,c,z	Applies a block reflector or its transpose/conjugate-transpose as returned by `p?tzrzf` to a general matrix.
p?larzc	c,z	Applies (multiplies by) the conjugate transpose of an elementary reflector as returned by `p?tzrzf` to a general matrix.
p?larzt	s,d,c,z	Forms the triangular factor T of a block reflector $H=I-VTV^H$ as returned by `p?tzrzf`.
p?lascl	s,d,c,z	Multiplies a general rectangular matrix by a real scalar defined as c_{to}/c_{from} .
p?laset	s,d,c,z	Initializes the off-diagonal elements of a matrix to α and the diagonal elements to β.
p?lasmsub	s,d	Looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero.
p?lassq	s,d,c,z	Updates a sum of squares represented in scaled form.
p?laswp	s,d,c,z	Performs a series of row interchanges on a general rectangular matrix.
p?latra	s,d,c,z	Computes the trace of a general square distributed matrix.
p?latrd	s,d,c,z	Reduces the first *nb* rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.
p?latrz	s,d,c,z	Reduces an upper trapezoidal matrix to upper triangular form by means of orthogonal/unitary transformations.
p?lauu2	s,d,c,z	Computes the product UU^H or L^HL, where U and L are upper or lower triangular matrices (local unblocked algorithm).
p?lauum	s,d,c,z	Computes the product UU^H or L^HL, where U and L are upper or lower triangular matrices.
p?lawil	s,d	Forms the Wilkinson transform.
p?org2l/p?ung2l	s,d,c,z	Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by `p?geqlf` (unblocked algorithm).

Table 7-1 ScaLAPACK Auxiliary Routines (continued)

Routine Name	Data Types	Description
p?org2r/p?ung2r	s,d,c,z	Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by p?geqrf (unblocked algorithm).
p?orgl2/p?ungl2	s,d,c,z	Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by p?gelqf (unblocked algorithm).
p?orgr2/p?ungr2	s,d,c,z	Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by p?gerqf (unblocked algorithm).
p?orm2l/p?unm2l	s,d,c,z	Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by p?geqlf (unblocked algorithm).
p?orm2r/p?unm2r	s,d,c,z	Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by p?geqrf (unblocked algorithm).
p?orml2/p?unml2	s,d,c,z	Multiplies a general matrix by the orthogonal/unitary matrix from an LQ factorization determined by p?gelqf (unblocked algorithm).
p?ormr2/p?unmr2	s,d,c,z	Multiplies a general matrix by the orthogonal/unitary matrix from an RQ factorization determined by p?gerqf (unblocked algorithm).
p?pbtrsv	s,d,c,z	Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a banded matrix computed by p?pbtrf.
p?pttrsv	s,d,c,z	Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a tridiagonal matrix computed by p?pttrf.
p?potf2	s,d,c,z	Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (local unblocked algorithm).
p?rscl	s,d,cs, zd	Multiplies a vector by the reciprocal of a real scalar.
p?sygs2/p?hegs2	s,d,c,z	Reduces a symmetric/Hermitian definite generalized eigenproblem to standard form, using the factorization results obtained from p?potrf (local unblocked algorithm).
p?sytd2/p?hetd2	s,d,c,z	Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (local unblocked algorithm).

Table 7-1 ScaLAPACK Auxiliary Routines (continued)

Routine Name	Data Types	Description
p?trti2	s,d,c,z	Computes the inverse of a triangular matrix (local unblocked algorithm).
?lamsh	s,d	Sends multiple shifts through a small (single node) matrix to maximize the number of bulges that can be sent through.
?laref	s,d	Applies Householder reflectors to matrices on either their rows or columns.
?lasorte	s,d	Sorts eigenpairs by real and complex data types.
?lasrt2	s,d	Sorts numbers in increasing or decreasing order.
?stein2	s,d	Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix, using inverse iteration.
?dbtf2	s,d,c,z	Computes an LU factorization of a general band matrix with no pivoting (local unblocked algorithm).
?dbtrf	s,d,c,z	Computes an LU factorization of a general band matrix with no pivoting (local blocked algorithm).
?dttrf	s,d,c,z	Computes an LU factorization of a general tridiagonal matrix with no pivoting (local blocked algorithm).
?dttrsv	s,d,c,z	Solves a general tridiagonal system of linear equations using the LU factorization computed by ?dttrf.
?pttrsv	s,d,c,z	Solves a symmetric (Hermitian) positive-definite tridiagonal system of linear equations, using the LDL^H factorization computed by ?pttrf.
?steqr2	s,d	Computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method.

p?lacgv

Conjugates a complex vector.

Syntax

```
call pclacgv (n, x, ix, jx, descx, incx)
call pzlacgv (n, x, ix, jx, descx, incx)
```

Description

The routine conjugates a complex vector of length *n*, sub(*x*), where sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx* = *descx*(*m_*) and *X*(*ix*:*ix*+*n*-1, *jx*) if *incx* = 1.

Input Parameters

n (global) INTEGER. The length of the distributed vector sub(*x*).

x (local).
COMPLEX for pclacgv
COMPLEX*16 for pzlacgv.
Pointer into the local memory to an array of DIMENSION (*lld_x*, *). On entry the vector to be conjugated *x*(*i*) = *X*(*ix*+(*jx*-1)**m_x* +(*i*-1)**incx*), $1 \le i \le n$.

ix (global) INTEGER. The row index in the global array *x* indicating the first row of sub(*x*).

jx (global) INTEGER. The column index in the global array *x* indicating the first column of sub(*x*).

descx (global and local) INTEGER.
Array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *X*.

incx (global) INTEGER. The global increment for the elements of *X*. Only two values of *incx* are supported in this version, namely 1 and *m_x*. *incx* must not be zero.

Output Parameters

x (local). On exit the conjugated vector.

p?max1

Finds the index of the element whose real part has maximum absolute value (similar to the Level 1 PBLAS p?amax*, but using the absolute value to the real part).*

Syntax

```
call pcmax1 (n, amax, indx, x, ix, jx, descx, incx)
call pzmax1 (n, amax, indx, x, ix, jx, descx, incx)
```

Description

This routine computes the global index of the maximum element in absolute value of a distributed vector sub*(x)*. The global index is returned in *indx* and the value is returned in *amax*, where sub*(x)* denotes $X(ix{:}ix+n-1, jx)$ if *incx* = 1,
$X(ix, jx{:}jx+n-1)$ if *incx* = *m_x*.

Input Parameters

n (global) pointer to `INTEGER`.
The number of components of the distributed vector sub*(x)*. $n \geq 0$.

x (local)
`COMPLEX` for `pcmax1`.
`COMPLEX*16` for `pzmax1`
Array containing the local pieces of a distributed matrix of dimension of at least `( (jx-1)*m_x + ix + ( n - 1 )*abs( incx ) )`. This array contains the entries of the distributed vector sub *(x)*.

ix (global) `INTEGER`. The row index in the global array *X* indicating the first row of sub*(x)*.

jx (global) `INTEGER`. The column index in the global array *X* indicating the first column of sub*(x)*

descx (global and local) `INTEGER`.
Array, `DIMENSION` (*dlen_*). The array descriptor for the distributed matrix *X*.

incx (global) `INTEGER`. The global increment for the elements of *X*. Only two values of *incx* are supported in this version, namely 1 and *m_x*. *incx* must not be zero.

Output Parameters

amax (global output) pointer to `REAL`. The absolute value of the largest entry of the distributed vector sub*(x)* only in the scope of sub*(x)*.

indx (global output) pointer to `INTEGER`. The global index of the element of the distributed vector sub*(x)* whose real part has maximum absolute value.

?combamax1

Finds the element with maximum real part absolute value and its corresponding global index.

Syntax

```
call ccombamax1 (v1, v2)
call zcombamax1 (v1, v2)
```

Description

This routine finds the element having maximum real part absolute value as well as its corresponding global index.

Input Parameters

v1 (local)
`COMPLEX` for `ccombamax1`
`COMPLEX*16` for `zcombamax1`
Array, `DIMENSION` 2.
The first maximum absolute value element and its global index. *v1*(1) = `amax`, *v1*(2) = `indx`.

v2 (local)
`COMPLEX` for `ccombamax1`
`COMPLEX*16` for `zcombamax1`
Array, `DIMENSION` 2.
The second maximum absolute value element and its global index. *v21*(1) = `amax`, *v2*(2) = `indx`.

Output Parameters

v1 (local). The first maximum absolute value element and its global index.
v1(1) = `amax`,
v1(2) = `indx`.

p?sum1

Forms the 1-norm of a complex vector similar to Level 1 PBLAS `p?asum`*, but using the true absolute value.*

Syntax

```
call pscsum1 (n, asum, x, ix, jx, descx, incx)
call pdzsum1 (n, asum, x, ix, jx, descx, incx)
```

Description

This routine returns the sum of absolute values of a complex distributed vector sub(*x*) in *asum*,

where sub(*x*) denotes *X*(*ix*:*ix*+*n*-1, *jx*:*jx*), if *incx* = 1,
X(*ix*:*ix*, *jx*:*jx*+*n*-1), if *incx* = *m_x*.

Based on `p?asum` from the Level 1 PBLAS. The change is to use the 'genuine' absolute value.

Input Parameters

n — (global) pointer to `INTEGER`.
The number of components of the distributed vector sub(*x*). $n \geq 0$.

x — (local)
`COMPLEX` for `pscsum1`
`COMPLEX*16` for `pdzsum1`.
Array containing the local pieces of a distributed matrix of dimension of at least `( (jx-1)*m_x + ix + ( n - 1 )*abs( incx ) )`. This array contains the entries of the distributed vector sub (*x*).

ix — (global) `INTEGER`. The row index in the global array *X* indicating the first row of sub(*x*).

jx — (global) `INTEGER`. The column index in the global array *X* indicating the first column of sub(*x*)

descx — (global and local) `INTEGER`.
Array, `DIMENSION 8`. The array descriptor for the distributed matrix *X*.

incx — (global) `INTEGER`. The global increment for the elements of *X*. Only two values of *incx* are supported in this version, namely 1 and *m_x*.

Output Parameters

asum (local) Pointer to `REAL`.
The sum of absolute values of the distributed vector sub(*x*) only in its scope.

p?dbtrsv

Computes an LU factorization of a general tridiagonal matrix with no pivoting. The routine is called by `p?dbtrs`.

Syntax

```
call psdbtrsv  (uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af,
    laf, work, lwork, info)
call pddbtrsv (uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af,
    laf, work, lwork, info)
call pcdbtrsv (uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af,
    laf, work, lwork, info)
call pzdbtrsv (uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af,
    laf, work, lwork, info)
```

Description

This routines solves a banded triangular system of linear equations

$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs)$ or

$A(1:n, ja:ja+n-1)^T * X = B(ib:ib+n-1, 1:nrhs)$ (for real flavors);
$A(1:n, ja:ja+n-1)^H * X = B(ib:ib+n-1, 1:nrhs)$ (for complex flavors),

where $A(1:n, ja:ja+n-1)$ is a banded triangular matrix factor produced by the Gaussian elimination code `PD@(dom_pre)BTRF` and is stored in $A(1:n, ja:ja+n-1)$ and *af*. The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to *uplo*, and the choice of solving $A(1:n, ja:ja+n-1)$ or $A(1:n, ja:ja+n-1)^T$ is dictated by the user by the parameter *trans*.

Routine `p?dbtrf` must be called first.

Input Parameters

uplo (global) CHARACTER.
If *uplo*= 'U', the upper triangle of A(1:*n*, *ja*:*ja*+*n*-1) is stored,
if *uplo* = 'L', the lower triangle of A(1:*n*, *ja*:*ja*+*n*-1) is stored.

trans (global) CHARACTER.

If *trans* ='N', solve with A(1:*n*, *ja*:*ja*+*n*-1),
if *trans* ='C', solve with conjugate transpose A(1:*n*, *ja*:*ja*+*n*-1).

n (global) INTEGER. The order of the distributed submatrix A; ($n \geq 0$).

bwl (global) INTEGER.
Number of subdiagonals. $0 \leq bwl \leq n-1$.

bwu (global) INTEGER.
Number of subdiagonals. $0 \leq bwu \leq n-1$.

nrhs (global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix B ($nrhs \geq 0$).

a (local).
REAL for psdbtrsv
DOUBLE PRECISION for pddbtrsv
COMPLEX for pcdbtrsv
COMPLEX*16 for pzdbtrsv.
Pointer into the local memory to an array with first DIMENSION
lld_a ≥(*bwl*+*bwu*+1) (stored in *desca*). On entry, this array contains the local pieces of the *n*-by-n unsymmetric banded distributed Cholesky factor L or $L^T A$(1:*n*, *ja*:*ja*+*n*-1).
This local portion is stored in the packed banded format used in LAPACK. Please see the *Application Notes* below and the ScaLAPACK manual for more detail on the format of distributed matrices.

ja (global) INTEGER. The index in the global array *a* that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
if 1*d* type (*dtype_a* = 501 or 502), *dlen* ≥ 7;
if 2*d* type (*dtype_a* = 1), *dlen* ≥ 9.
The array descriptor for the distributed matrix A. Contains information of mapping of A to memory.

b (local)
REAL for psdbtrsv
DOUBLE PRECISION for pddbtrsv
COMPLEX for pcdbtrsv
COMPLEX*16 for pzdbtrsv.
Pointer into the local memory to an array of local lead DIMENSION *lld_b* ≥ *nb*. On entry, this array contains the local pieces of the right hand sides *B*(*ib*:*ib*+*n*-1, 1:*nrhs*).

ib (global) INTEGER. The row index in the global array *b* that points to the first row of the matrix to be operated on (which may be either all of *b* or a submatrix of *B*).

desb (global and local) INTEGER array of DIMENSION (*dlen_*).
if 1*d* type (*dtype_b* =502), *dlen* ≥7;
if 2*d* type (*dtype_b* =1), *dlen* ≥ 9.
The array descriptor for the distributed matrix *B*. Contains information of mapping *B* to memory.

laf (local) INTEGER. Size of user-input Auxiliary Filling space *af*.
laf must be ≥ *nb**(*bwl*+*bwu*)+6*max(*bwl*, *bwu*)*max(*bwl*, *bwu*). If *laf* is not large enough, an error code is returned and the minimum acceptable size will be returned in *af*(1).

work (local).

REAL for psdbtrsv
DOUBLE PRECISION for pddbtrsv
COMPLEX for pcdbtrsv
COMPLEX*16 for pzdbtrsv.
Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.

lwork (local or global) INTEGER.
Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.
lwork ≥ max(*bwl*, *bwu*)**nrhs*.

Output Parameters

a (local).
This local portion is stored in the packed banded format used in LAPACK. Please see the *Application Notes* below and the ScaLAPACK manual for more detail on the format of distributed matrices.

b On exit, this contains the local piece of the solutions distributed matrix *X*.

a (local).

REAL for psdbtrsv
DOUBLE PRECISION for pddbtrsv
COMPLEX for pcdbtrsv
COMPLEX*16 for pzdbtrsv.
Auxiliary Filling Space. Filling is created during the factorization routine p?dbtrf and this is stored in *af*. If a linear system is to be solved using p?dbtrf after the factorization routine, *af* must not be altered after the factorization.

work On exit, *work*(1) contains the minimal *lwork*.

info (local).INTEGER. If *info* = 0, the execution is successful.
< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?dttrsv

Computes an LU factorization of a general band matrix, using partial pivoting with row interchanges. The routine is called by p?dttrs.

Syntax

```
call psdttrsv (uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af,
    laf, work, lwork, info)
call pddttrsv (uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af,
    laf, work, lwork, info)
call pcdttrsv (uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af,
    laf, work, lwork, info)
call pzdttrsv (uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af,
    laf, work, lwork, info)
```

Description

This routine solves a tridiagonal triangular system of linear equations

$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs)$ or

$A(1:n, ja:ja+n-1)^T * X = B(ib:ib+n-1, 1:nrhs)$ for real flavors;
$A(1:n, ja:ja+n-1)^H * X = B(ib:ib+n-1, 1:nrhs)$ for complex flavors,

where $A(1:n, ja:ja+n-1)$ is a tridiagonal matrix factor produced by the Gaussian elimination code PS@(dom_pre)TTRF and is stored in $A(1:n, ja:ja+n-1)$ and *af*.

The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to *uplo*, and the choice of solving $A(1:n, ja:ja+n-1)$ or $A(1:n, ja:ja+n-1)^T$ is dictated by the user by the parameter *trans*.

Routine p?dttrf must be called first.

Input Parameters

uplo (global) CHARACTER.
If *uplo*= 'U', the upper triangle of $A(1:n, ja:ja+n-1)$ is stored,
if *uplo* = 'L', the lower triangle of $A(1:n, ja:ja+n-1)$ is stored.

trans (global) CHARACTER.

If *trans* ='N', solve with $A(1:n, ja:ja+n-1)$,
if *trans* ='C', solve with conjugate transpose $A(1:n, ja:ja+n-1)$.

n (global) INTEGER. The order of the distributed submatrix *A*; ($n \geq 0$).

nrhs (global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix $B(ib:ib+n-1, 1:nrhs)$. ($nrhs \geq 0$).

dl (local).
REAL for psdttrsv
DOUBLE PRECISION for pddttrsv
COMPLEX for pcdttrsv
COMPLEX*16 for pzdttrsv.
Pointer to local part of global vector storing the lower diagonal of the matrix.
Globally, *dl*(1) is not referenced, and *dl* must be aligned with *d*.
Must be of size ≥ *desca*(*nb_*).

d (local).
REAL for psdttrsv
DOUBLE PRECISION for pddttrsv
COMPLEX for pcdttrsv
COMPLEX*16 for pzdttrsv.
Pointer to local part of global vector storing the main diagonal of the matrix.

du (local).
REAL for psdttrsv
DOUBLE PRECISION for pddttrsv
COMPLEX for pcdttrsv
COMPLEX*16 for pzdttrsv.
Pointer to local part of global vector storing the upper diagonal of the matrix. Globally, *du*(*n*) is not referenced, and *du* must be aligned with *d*.

ja (global) INTEGER. The index in the global array *a* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local). INTEGER array of DIMENSION (*dlen_*).
if 1*d* type (*dtype_a* = 501 or 502), *dlen* ≥ 7;
if 2*d* type (*dtype_a* = 1), *dlen* ≥ 9.
The array descriptor for the distributed matrix *A*. Contains information of mapping of *A* to memory.

b (local)
REAL for psdttrsv
DOUBLE PRECISION for pddttrsv
COMPLEX for pcdttrsv
COMPLEX*16 for pzdttrsv.
Pointer into the local memory to an array of local lead DIMENSION *lld_b* ≥ *nb*. On entry, this array contains the local pieces of the right hand sides *B*(*ib*:*ib*+*n*-1, 1:*nrhs*).

ib (global).INTEGER. The row index in the global array *b* that points to the first row of the matrix to be operated on (which may be either all of *b* or a submatrix of *B*).

desb (global and local).INTEGER array of DIMENSION (*dlen_*).
if 1*d* type (*dtype_b* = 502), *dlen* ≥7;
if 2*d* type (*dtype_b* = 1), *dlen* ≥ 9.
The array descriptor for the distributed matrix *B*. Contains information of mapping *B* to memory.

laf (local).INTEGER.Size of user-input Auxiliary Filling space *af*.
laf must be ≥ 2*(*nb*+2). If *laf* is not large enough, an error code is returned and the minimum acceptable size will be returned in *af*(1).

work (local).

REAL for psdttrsv
DOUBLE PRECISION for pddttrsv
COMPLEX for pcdttrsv

COMPLEX*16 for pzdttrsv.
Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.

lwork (local or global).INTEGER.
Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.
lwork ≥ 10**npcol*+4**nrhs*.

Output Parameters

dl (local).
On exit, this array contains information containing the factors of the matrix.

d On exit, this array contains information containing the factors of the matrix.
Must be of size ≥ *desca* (*nb_*).

b On exit, this contains the local piece of the solutions distributed matrix X.

af (local).

REAL for psdttrsv
DOUBLE PRECISION for pddttrsv
COMPLEX for pcdttrsv
COMPLEX*16 for pzdttrsv.
Auxiliary Filling Space. Filling is created during the factorization routine p?dttrf and this is stored in *af*. If a linear system is to be solved using p?dttrs after the factorization routine, *af* must not be altered after the factorization.

work On exit, *work*(1) contains the minimal *lwork*.

info (local).INTEGER.
If *info*=0, the execution is successful.
if *info*< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?gebd2

Reduces a general rectangular matrix to real bidiagonal form by an orthogonal/unitary transformation (unblocked algorithm).

Syntax

```
call psgebd2 (m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pdgebd2 (m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pcgebd2 (m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pzgebd2 (m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
```

Description

This routine reduces a real/complex general *m*-by-*n* distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) to upper or lower bidiagonal form *B* by an orthogonal/unitary transformation:

$Q' * \text{sub}(A) * P = B$.

If $m \geq n$, *B* is the upper bidiagonal; if $m < n$, *B* is the lower bidiagonal.

Input Parameters

m (global) INTEGER.
The number of rows of the distributed submatrix sub(*A*). ($m \geq 0$).

n (global) INTEGER. The order of the distributed submatrix sub(*A*). ($n \geq 0$).

a (local).
REAL for psgebd2
DOUBLE PRECISION for pdgebd2
COMPLEX for pcgebd2
COMPLEX*16 for pzgebd2.
Pointer into the local memory to an array of DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the general distributed matrix sub(*A*).

ia, ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local).
REAL for psgebd2
DOUBLE PRECISION for pdgebd2
COMPLEX for pcgebd2
COMPLEX*16 for pzgebd2.
This is a workspace array of DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least *lwork* ≥ max(*mpa0*, *nqa0*), where
nb = *mb_a* = *nb_a*, *iroffa* = mod(*ia*-1, *nb*)
iarow = indxg2p (*ia*, *nb*, *myrow*, *rsrc_a*, *nprow*),
iacol = indxg2p (*ja*, *nb*, *mycol*, *csrc_a*, *npcol*),
mpa0 = numroc(*m*+*iroffa*, *nb*, *myrow*, *iarow*, *nprow*),
nqa0 = numroc(*n*+*icoffa*, *nb*, *mycol*, *iacol*, *npcol*).

indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a (local).
On exit, if $m \geq n$, the diagonal and the first superdiagonal of sub(*A*) are overwritten with the upper bidiagonal matrix *B*; the elements below the diagonal, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and the elements above the first superdiagonal, with the array *taup*, represent the orthogonal matrix *P* as a product of elementary reflectors.
If $m < n$, the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix *B*; the elements below the first subdiagonal, with the array *tauq*, represent the orthogonal/unitary 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 *Applications Notes* below.

d (local)
`REAL` for `psgebd2`
`DOUBLE PRECISION` for `pdgebd2`
`COMPLEX` for `pcgebd2`
`COMPLEX*16` for `pzgebd2`.
Array, `DIMENSION` *LOCc*(*ja*+min(*m*,*n*)-1) if $m \geq n$; *LOCr*(*ia*+min(*m*, *n*)-1) otherwise. The distributed diagonal elements of the bidiagonal matrix *B*: *d*(*i*) = *a*(*i*,*i*). *d* is tied to the distributed matrix *A*.

e (local)
`REAL` for `psgebd2`
`DOUBLE PRECISION` for `pdgebd2`
`COMPLEX` for `pcgebd2`
`COMPLEX*16` for `pzgebd2`.
Array, `DIMENSION` *LOCc*(*ja*+min(*m*,*n*)-1) if $m \geq n$; *LOCr*(*ia*+min(*m*,*n*)-2) otherwise. The distributed diagonal elements of the bidiagonal matrix *B*:
if $m \geq n$, *e*(*i*) = *a*(*i*, *i*+1) for *i* = 1, 2, ... , *n*-1;
if $m \geq n$, *e*(*i*) = *a*(*i*+1, *i*) for *i* = 1, 2, ..., *m*-1. *e* is tied to the distributed matrix *A*.

tauq (local).
`REAL` for `psgebd2`
`DOUBLE PRECISION` for `pdgebd2`
`COMPLEX` for `pcgebd2`
`COMPLEX*16` for `pzgebd2`.
Array, `DIMENSION` *LOCc*(*ja*+min(*m*,*n*)-1). The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q*. *tauq* is tied to the distributed matrix *A*.

taup (local).
`REAL` for `psgebd2`
`DOUBLE PRECISION` for `pdgebd2`
`COMPLEX` for `pcgebd2`
`COMPLEX*16` for `pzgebd2`.
Array, `DIMENSION` *LOCr*(*ia*+min(*m*,*n*)-1). The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *P*. *taup* is tied to the distributed matrix *A*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local) INTEGER.
If *info* = 0, the execution is successful.
if *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrices Q and P are represented as products of elementary reflectors:

If $m \geq n$,

$Q = H(1)\, H(2) \ldots H(n)$ and $P = G(1)\, G(2) \ldots G(n-1)$

Each $H(i)$ and $G(i)$ has the form:

$H(i) = I - tauq * v * v'$ and $G(i) = I - taup * u * u'$,

where *tauq* and *taup* are real/complex scalars, and *v* and *u* are real/complex vectors.
$v(1:i-1) = 0$, $v(i) = 1$, and $v(i+i:m)$ is stored on exit in

$A(ia+i-ia+m-1, ja+i-1)$;
$u(1:i) = 0$, $u(i+1) = 1$, and $u(i+2:n)$ is stored on exit in
$A(ia+i-1, ja+i+1:ja+n-1)$;
tauq is stored in $TAUQ(ja+i-1)$ and *taup* in $TAUP(ia+i-1)$.

If $m < n$,

$v(1:i) = 0$, $v(i+1) = 1$, and $v(i+2:m)$ is stored on exit in
$A(ia+i+1:ia+m-1, ja+i-1)$;
$u(1:i-1) = 0$, $u(i) = 1$, and $u(i+1:n)$ is stored on exit in
$A(ia+i-1, ja+i:ja+n-1)$;
tauq is stored in $TAUQ(ja+i-1)$ and *taup* in $TAUP(ia+i-1)$.

The contents of sub(A) on exit are illustrated by the following examples:

m = 6 and n = 5 ($m > n$):

$$\begin{bmatrix} d & e & u1 & u1 & u1 \\ v1 & d & e & u2 & u2 \\ v1 & v2 & d & e & u3 \\ v1 & v2 & v3 & d & e \\ v1 & v2 & v3 & v4 & d \\ v1 & v2 & v3 & v4 & v5 \end{bmatrix}$$

m = 5 and n = 6 ($m < n$):

$$\begin{bmatrix} d & u1 & u1 & u1 & u1 & u1 \\ e & d & u2 & u2 & u2 & u2 \\ v1 & e & d & u3 & u3 & u3 \\ v1 & v2 & e & d & u4 & u4 \\ v1 & v2 & v3 & e & d & u5 \end{bmatrix}$$

where d and e denote diagonal and off-diagonal elements of B, `vi` denotes an element of the vector defining H(`i`), and `ui` an element of the vector defining G(`i`).

p?gehd2

Reduces a general matrix to upper Hessenberg form by an orthogonal/unitary similarity transformation (unblocked algorithm).

Syntax

```
call psgehd2 (n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pdgehd2 (n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pcgehd2 (n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pzgehd2 (n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
```

Description

This routine reduces a real/complex general distributed matrix sub(A) to upper Hessenberg form H by an orthogonal/unitary similarity transformation: $Q' * \text{sub}(A) * Q = H$, where sub($A$) = $A(ia+n-1:ia+n-1, ja+n-1:ja+n-1)$.

Input Parameters

n (global) `INTEGER`. The order of the distributed submatrix A. ($n \geq 0$).

ilo, ihi (global) INTEGER. It is assumed that sub(*A*) is already upper triangular in rows *ia*:*ia*+*ilo*-2 and *ia*+*ihi*:*ia*+*n*-1 and columns *ja*:*ja*+*jlo*-2 and *ja*+*jhi*:*ja*+*n*-1. See *Application Notes* for further information.
If $n > 0$, $1 \le ilo \le ihi \le n$; otherwise set *ilo* = 1, *ihi* = *n*.

a (local).

REAL for psgehd2
DOUBLE PRECISION for pdgehd2
COMPLEX for pcgehd2
COMPLEX*16 for pzgehd2.
Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the *n*-by-*n* general distributed matrix sub(*A*) to be reduced.

ia,ja (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local).
REAL for psgehd2
DOUBLE PRECISION for pdgehd2
COMPLEX for pcgehd2
COMPLEX*16 for pzgehd2.
This is a workspace array of DIMENSION (*lwork*).

lwork (local or global). INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least $lwork \ge nb + \max(npa0, nb)$, where
$nb = mb\_a = nb\_a$, $iroffa = \mathrm{mod}(ia-1, nb)$
iarow = indxg2p (*ia, nb, myrow, rsrc_a, nprow*),
npa0 = numroc (*ihi+iroffa, nb, myrow, iarow, nprow*).

indxg2p and numroc are ScaLAPACK tool functions;
myrow, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a (local).On exit, the upper triangle and the first subdiagonal of sub(*A*) are overwritten with the upper Hessenberg matrix *H*, and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors. See *Application Notes* below.

tau (local).
REAL for psgehd2
DOUBLE PRECISION for pdgehd2
COMPLEX for pcgehd2
COMPLEX*16 for pzgehd2.
Array, DIMENSION *LOCc*(*ja*+*n*-2) The scalar factors of the elementary reflectors (see *Application Notes* below). Elements *ja*:*ja*+*ilo*-2 and *ja*+*ihi*:*ja*+*n*-2 of *tau* are set to zero. *tau* is tied to the distributed matrix *A*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local).INTEGER.
If *info* = 0, the execution is successful.
if *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of (*ihi*-*ilo*) elementary reflectors

Q = *H*(*ilo*) *H*(*ilo*+1) . . . *H*(*ihi*-1).

Each *H*(*i*) has the form

H(*i*) = *I* - *tau* **v* **v*',

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1:*i*) = 0, *v*(*i*+1) = 1 and *v*(*ihi*+1:*n*) = 0; *v*(*i*+2:*ihi*) is stored on exit in *A*(*ia*+*ilo*+*i*:*ia*+*ihi*-1, *ia*+*ilo*+*i*-2), and *tau* in *tau*(*ja*+*ilo*+*i*-2).

The contents of A(`ia:ia+n-1, ja:ja+n-1`) are illustrated by the following example, with $n = 7$, *`ilo`* = 2 and *`ihi`* = 6:

on entry

$$\begin{bmatrix} a & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & & & & & & a \end{bmatrix}$$

on exit

$$\begin{bmatrix} a & a & h & h & h & h & a \\ & a & h & h & h & h & a \\ & h & h & h & h & h & h \\ & v2 & h & h & h & h & h \\ & v2 & v3 & h & h & h & h \\ & v2 & v3 & v4 & h & h & h \\ & & & & & & a \end{bmatrix}$$

where *`a`* denotes an element of the original matrix sub(A), *`h`* denotes a modified element of the upper Hessenberg matrix H, and *`vi`* denotes an element of the vector defining H(`ja+ilo+i-2`).

p?gelq2

Computes an LQ factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
call psgelq2 (m, n, a, ia, ja, desca, tau, work, lwork, info)
call psgelq2 (m, n, a, ia, ja, desca, tau, work, lwork, info)
call psgelq2 (m, n, a, ia, ja, desca, tau, work, lwork, info)
call psgelq2 (m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

This routine computes an LQ factorization of a real/complex distributed *m*-by-*n* matrix sub(A) = A(`ia:ia+m-1, ja:ja+n-1`) = $L * Q$.

Input Parameters

m (global) `INTEGER.`
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(A). ($m \geq 0$).

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). ($n \geq 0$).

a (local).
REAL for psgelq2
DOUBLE PRECISION for pdgelq2
COMPLEX for pcgelq2
COMPLEX*16 for pzgelq2.
Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the *m*-by-*n* distributed matrix sub(*A*) which is to be factored.

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local).
REAL for psgelq2
DOUBLE PRECISION for pdgelq2
COMPLEX for pcgelq2
COMPLEX*16 for pzgelq2.
This is a workspace array of DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least *lwork* ≥ *nq0* + max(1, *mp0*), where

iroff = mod(*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),
iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),
mp0 = numroc(*m*+*iroff*, *mb_a*, *myrow*, *iarow*, *nprow*),
nq0 = numroc(*n*+*icoff*, *nb_a*, *mycol*, *iacol*, *npcol*),

indxg2p and numroc are ScaLAPACK tool functions;
myrow, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a (local).
On exit, the elements on and below the diagonal of sub(*A*) contain the *m* by min(*m*, *n*) lower trapezoidal matrix *L* (*L* is lower triangular if $m \le n$); the elements above the diagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below).

tau (local).
`REAL` for `psgelq2`
`DOUBLE PRECISION` for `pdgelq2`
`COMPLEX` for `pcgelq2`
`COMPLEX*16` for `pzgelq2`.
Array, `DIMENSION` *LOCr*(*ia*+min(*m*, *n*)-1). This array contains the scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local).`INTEGER`.
If *info* = 0, the execution is successful.
if *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*ia*+*k*-1) *H*(*ia*+*k*-2) . . . *H*(*ia*) for real flavors,
Q = *H*(*ia*+*k*-1)' *H*(*ia*+*k*-2)' . . . *H*(*ia*)' for complex flavors,

where *k* = min(*m*, *n*).

Each *H*(*i*) has the form

H(*i*) = *I* - *tau* * *v* * *v'*

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1:*i*-1) = 0 and *v*(*i*) = 1; *v*(*i*+1:*n*) (for real flavors) or `conjg`(*v*(*i*+1:*n*)) (for complex flavors) is stored on exit in *A*(*ia*+*i*-1,*ja*+*i*:*ja*+*n*-1), and *tau* in *TAU*(*ia*+*i*-1).

p?geql2

Computes a QL factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
call psgeql2 (m, n, a, ia, ja, desca, tau, work, lwork, info)
call psgeql2 (m, n, a, ia, ja, desca, tau, work, lwork, info)
call psgeql2 (m, n, a, ia, ja, desca, tau, work, lwork, info)
call psgeql2 (m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine computes a *QL* factorization of a real/complex distributed *m*-by-*n* matrix sub(A) = A(ia:ia+m-1, ja:ja+n-1) = Q *L.

Input Parameters

m (global) `INTEGER.`
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(A). ($m \geq 0$).

n (global) `INTEGER`.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(A). ($n \geq 0$).

a (local).
`REAL` for `psgeql2`
`DOUBLE PRECISION` for `pdgeql2`
`COMPLEX` for `pcgeql2`
`COMPLEX*16` for `pzgeql2`.
Pointer into the local memory to an array of `DIMENSION`(*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the *m*-by-*n* distributed matrix sub(A) which is to be factored.

ia, ja (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix A, respectively.

desca (global and local) `INTEGER` array, `DIMENSION` (*dlen_*). The array descriptor for the distributed matrix A.

work (local).
REAL for psgeql2
DOUBLE PRECISION for pdgeql2
COMPLEX for pcgeql2
COMPLEX*16 for pzgeql2.
This is a workspace array of DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least $lwork \geq mp0 + \max(1, nq0)$, where
iroff = mod(*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),
iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),
mp0 = numroc(*m*+*iroff*, *mb_a*, *myrow*, *iarow*, *nprow*),
nq0 = numroc(*n*+*icoff*, *nb_a*, *mycol*, *iacol*, *npcol*),

indxg2p and numroc are ScaLAPACK tool functions;
myrow, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a (local).
On exit, if $m \geq n$, the lower triangle of the distributed submatrix *A*(*ia*+*m*-*n*:*ia*+*m*-1, *ja*:*ja*+*n*-1) contains the *n*-by-*n* lower triangular matrix *L*; if $m \leq n$, the elements on and below the (*n*-*m*)-th superdiagonal contain the *m* by *n* lower trapezoidal matrix *L*; the remaining elements, with the array *tau*, represent the orthogonal/ unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below).

tau (local).
REAL for psgeql2
DOUBLE PRECISION for pdgeql2
COMPLEX for pcgeql2
COMPLEX*16 for pzgeql2.
Array, DIMENSION *LOCc*(*ja*+*n*-1). This array contains the scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local).INTEGER.
If *info* = 0, the execution is successful.
if *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*ja*+*k*-1) . . . *H*(*ja*+1) *H*(*ja*), where *k* = min(*m*, *n*).

Each *H*(*i*) has the form

H(*i*) = *I* - *tau* * *v* * *v'*

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(*m*-*k*+*i*+1:*m*) = 0 and *v*(*m*-*k*+*i*) = 1; *v*(1:*m*-*k*+*i*-1) is stored on exit in *A*(*ia*:*ia*+*m*-*k*+*i*-2, *ja*+*n*-*k*+*i*-1), and *tau* in *TAU*(*ja*+*n*-*k*+*i*-1).

p?geqr2

Computes a QR factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
call psgeqr2 (m, n, a, ia, ja, desca, tau, work, lwork, info)
call psgeqr2 (m, n, a, ia, ja, desca, tau, work, lwork, info)
call psgeqr2 (m, n, a, ia, ja, desca, tau, work, lwork, info)
call psgeqr2 (m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

This routine computes a *QR* factorization of a real/complex distributed *m*-by-*n* matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) = *Q* **R*.

Input Parameters

m (global). INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). ($m \geq 0$).

n (global).INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). ($n \geq 0$).

a (local).
REAL for psgeqr2
DOUBLE PRECISION for pdgeqr2
COMPLEX for pcgeqr2
COMPLEX*16 for pzgeqr2.
Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the *m*-by-*n* distributed matrix sub(*A*) which is to be factored.

ia,ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local).
REAL for psgeqr2
DOUBLE PRECISION for pdgeqr2
COMPLEX for pcgeqr2
COMPLEX*16 for pzgeqr2.
This is a workspace array of DIMENSION (*lwork*).

lwork (local or global). INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least *lwork* ≥ *mp0* + max(1, *nq0*), where
iroff = mod(*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia, mb_a, myrow, rsrc_a, nprow*),
iacol = indxg2p(*ja, nb_a, mycol, csrc_a, npcol*),
mp0 = numroc(*m+iroff, mb_a, myrow, iarow, nprow*),
nq0 = numroc(*n+icoff, nb_a, mycol, iacol, npcol*),

indxg2p and numroc are ScaLAPACK tool functions;
myrow, mycol, nprow, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a (local).
On exit, the elements on and above the diagonal of sub(A) contain the min(m, n) by n upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array *tau*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see *Application Notes* below).

tau (local).
REAL for psgeqr2
DOUBLE PRECISION for pdgeqr2
COMPLEX for pcgeqr2
COMPLEX*16 for pzgeqr2.
Array, DIMENSION $LOCc(ja+\min(m,n)-1)$. This array contains the scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix A.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local).INTEGER.
If *info* = 0, the execution is successful.
if *info* < 0:

If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$Q = H(ja)\, H(ja+1) \ldots H(ja+k-1)$, where $k = \min(m,n)$.

Each $H(i)$ has the form

$H(j) = I - tau * v * v'$,

where *tau* is a real/complex scalar, and *v* is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$, and *tau* in $TAU(ja+i-1)$.

p?gerq2

Computes an RQ factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
call psgerq2 (m, n, a, ia, ja, desca, tau, work, lwork, info)
call psgerq2 (m, n, a, ia, ja, desca, tau, work, lwork, info)
call psgerq2 (m, n, a, ia, ja, desca, tau, work, lwork, info)
call psgerq2 (m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

This routine computes an *RQ* factorization of a real/complex distributed *m*-by-*n* matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) = *R***Q*.

Input Parameters

m (global). `INTEGER`.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). ($m \geq 0$).

n (global).`INTEGER`.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). ($n \geq 0$).

a (local).
`REAL` for `psgerq2`
`DOUBLE PRECISION` for `pdgerq2`
`COMPLEX` for `pcgerq2`
`COMPLEX*16` for `pzgerq2`.
Pointer into the local memory to an array of `DIMENSION`(*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the *m*-by-*n* distributed matrix sub(*A*) which is to be factored.

ia, *ja* (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) `INTEGER` array, `DIMENSION` (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local).
REAL for psgerq2
DOUBLE PRECISION for pdgerq2
COMPLEX for pcgerq2
COMPLEX*16 for pzgerq2.
This is a workspace array of DIMENSION (*lwork*).

lwork (local or global). INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least *lwork* ≥ *nq0* + max(1, *mp0*), where

iroff = mod (*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),
iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),
mp0 = numroc(*m*+*iroff*, *mb_a*, *myrow*, *iarow*, *nprow*),
nq0 = numroc(*n*+*icoff*, *nb_a*, *mycol*, *iacol*, *npcol*),

indxg2p and numroc are ScaLAPACK tool functions;
myrow, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a (local).
On exit, if *m* ≤ *n*, the upper triangle of *A*(*ia*+*m*-*n*:*ia*+*m*-*1*, *ja*:*ja*+*n*-*1*) contains the *m*-by-*m* upper triangular matrix *R*; if *m* ≥ *n*, the elements on and above the (*m*-*n*)-th subdiagonal contain the *m* by *n* upper trapezoidal matrix *R*; the remaining elements, with the array *tau*, represent the orthogonal/ unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below).

tau (local).
REAL for psgeqr2
DOUBLE PRECISION for pdgeqr2
COMPLEX for pcgeqr2
COMPLEX*16 for pzgeqr2.
Array, DIMENSION *LOCr*(*ia*+*m* -1). This array contains the scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local).INTEGER.
If *info* = 0, the execution is successful.
if *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*ia*) *H*(*ia*+1) . . . *H*(*ia*+*k*-1) for real flavors,
Q = *H*(*ia*)' *H*(*ia*+1)' . . . *H*(*ia*+*k*-1)' for complex flavors,

where *k* = min(*m*, *n*).

Each *H*(*i*) has the form

H(*i*) = *I* - *tau* **v* **v'*,

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(*n*-*k*+*i*+1:*n*) = 0 and *v*(*n*-*k*+*i*) = 1; *v*(1:*n*-*k*+*i*-*1*) for real flavors or conjg(*v*(1:*n*-*k*+*i*-1)) for complex flavors is stored on exit in *A*(*ia*+*m*-*k*+*i*-1, *ja*:*ja*+*n*-*k*+*i*-2), and *tau* in *TAU*(*ia*+*m*-*k*+*i*-1).

p?getf2

Computes an LU factorization of a general matrix, using partial pivoting with row interchanges (local blocked algorithm).

Syntax

```
call psgetf2 (m, n, a, ia, ja, desca, ipiv, info)
call pdgetf2 (m, n, a, ia, ja, desca, ipiv, info)
call pcgetf2 (m, n, a, ia, ja, desca, ipiv, info)
call pzgetf2 (m, n, a, ia, ja, desca, ipiv, info)
```

Description

This routine computes an *LU* factorization of a general *m*-by-*n* distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) using partial pivoting with row interchanges.

The factorization has the form sub(A) = $P * L * U$, where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$), and U is upper triangular (upper trapezoidal if $m < n$). This is the right-looking Parallel Level 2 BLAS version of the algorithm.

Input Parameters

m (global). INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(A). ($m \geq 0$).

n (global). INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(A). (nb_a - mod(ja-1, nb_a) $\geq n \geq 0$).

a (local).
REAL for psgetf2
DOUBLE PRECISION for pdgetf2
COMPLEX for pcgetf2
COMPLEX*16 for pzgetf2.
Pointer into the local memory to an array of DIMENSION (lld_a, $LOCc$(ja+n-1)).
On entry, this array contains the local pieces of the m-by-n distributed matrix sub(A).

ia, ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix sub(A), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

Output Parameters

ipiv (local). INTEGER.
Array, DIMENSION ($LOCr$(m_a) + mb_a). This array contains the pivoting information. *ipiv*(*i*) -> The global row that local row *i* was swapped with. This array is tied to the distributed matrix A.

info (local). INTEGER.
If *info* = 0: successful exit.
If *info* < 0:

- if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (i*100+j),
- if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info* > 0: If *info* = *k*, *u*(*ia*+*k*-1, *ja*+*k*-1) is exactly zero. The factorization has been completed, but the factor *u* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

p?labrd

Reduces the first nb *rows and columns of a general rectangular matrix A to real bidiagonal form by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A.*

```
call pslabrd (m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y,
    iy, jy, descy, work)
call pdlabrd (m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y,
    iy, jy, descy, work)
call pclabrd (m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y,
    iy, jy, descy, work)
call pzlabrd (m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y,
    iy, jy, descy, work)
```

Description

This routine reduces the first *nb* rows and columns of a real/complex general *m*-by-*n* distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) to upper or lower bidiagonal form by an orthogonal/unitary transformation $Q' * A * P$, and returns the matrices *X* and *Y* necessary to apply the transformation to the unreduced part of sub(*A*).

If $m \geq n$, sub(*A*) is reduced to upper bidiagonal form;
if $m < n$, sub(*A*) is reduced to lower bidiagonal form.

This is an auxiliary routine called by p?gebrd.

Input Parameters

m (global). INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). ($m \geq 0$).

n (global).INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). ($n \geq 0$).

nb (global) INTEGER. The number of leading rows and columns of sub(*A*) to be reduced.

a (local).
REAL for pslabrd
DOUBLE PRECISION for pdlabrd
COMPLEX for pclabrd
COMPLEX*16 for pzlabrd
Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the general distributed matrix sub(*A*).

ia, ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

ix, jx (global) INTEGER. The row and column indices in the global array *x* indicating the first row and the first column of the submatrix sub(*X*), respectively.

descx (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *X*.

iy, jy (global) INTEGER. The row and column indices in the global array *y* indicating the first row and the first column of the submatrix sub(*Y*), respectively.

descy (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *Y*.

work (local).
REAL for pslabrd
DOUBLE PRECISION for pdlabrd
COMPLEX for pclabrd
COMPLEX*16 for pzlabrd
Workspace array, DIMENSION (*lwork*)
lwork ≥ *nb_a* + *nq*,
with *nq* = numroc(*n* + mod(*ia*-1, *nb_y*), *nb_y*, *mycol*, *iacol*, *npcol*)
iacol = indxg2p (*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*)

`indxg2p` and `numroc` are ScaLAPACK tool functions;
myrow, mycol, nprow, and *npcol* can be determined by calling the subroutine `blacs_gridinfo`.

Output Parameters

a (local)
On exit, the first *nb* rows and columns of the matrix are overwritten; the rest of the distributed matrix sub(*A*) is unchanged.
If $m \geq n$, elements on and below the diagonal in the first *nb* columns, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors; and elements above the diagonal in the first *nb* rows, with the array *taup*, represent the orthogonal/unitary matrix *P* as a product of elementary reflectors.

If $m < n$, elements below the diagonal in the first *nb* columns, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and elements on and above the diagonal in the first *nb* rows, with the array *taup*, represent the orthogonal/unitary matrix *P* as a product of elementary reflectors. See *Application Notes* below.

e (local).
`REAL` for `pslabrd`
`DOUBLE PRECISION` for `pdlabrd`
`COMPLEX` for `pclabrd`
`COMPLEX*16` for `pzlabrd`
Array, `DIMENSION` $LOCr(ia+\min(m,n)-1)$ if $m \geq n$; $LOCc(ja+\min(m,n)-2)$ otherwise. The distributed off-diagonal elements of the bidiagonal distributed matrix *B*:
if $m \geq n$, $E(i) = A(ia+i-1, ja+i)$ for $i = 1, 2, ..., n-1$;
if $m < n$, $E(i) = A(ia+i, ja+i-1)$ for $i = 1, 2, ..., m-1$.
E is tied to the distributed matrix *A*.

tauq, *taup* (local).
`REAL` for `pslabrd`
`DOUBLE PRECISION` for `pdlabrd`
`COMPLEX` for `pclabrd`
`COMPLEX*16` for `pzlabrd`
Array `DIMENSION` $LOCc(ja+\min(m, n)-1)$ for *tauq*, `DIMENSION` $LOCr(ia+\min(m, n)-1)$ for *taup*. The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q* for *tauq*, *P* for *taup*. *tauq* and *taup* are tied to the distributed matrix *A*. See *Application Notes* below.

x (local)
REAL for pslabrd
DOUBLE PRECISION for pdlabrd
COMPLEX for pclabrd
COMPLEX*16 for pzlabrd
Pointer into the local memory to an array of DIMENSION (*lld_x*, *nb*). On exit, the local pieces of the distributed *m*-by-*nb* matrix $X(ix:ix+m-1, jx:jx+nb-1)$ required to update the unreduced part of sub(*A*).

y (local).
REAL for pslabrd
DOUBLE PRECISION for pdlabrd
COMPLEX for pclabrd
COMPLEX*16 for pzlabrd
Pointer into the local memory to an array of DIMENSION (*lld_y*, *nb*). On exit, the local pieces of the distributed *n*-by-*nb* matrix $Y(iy:iy+n-1, jy:jy+nb-1)$ required to update the unreduced part of sub(*A*).

Application Notes

The matrices *Q* and *P* are represented as products of elementary reflectors:

$Q = H(1)\,H(2)\,...\,H(nb)$ and $P = G(1)\,G(2)\,...\,G(nb)$

Each $H(i)$ and $G(i)$ has the form:

$H(i) = I - tauq*v*v'$ and $G(i) = I - taup*u*u'$,

where *tauq* and *taup* are real/complex scalars, and *v* and *u* are real/complex vectors.

If $m \geq n$, $v(1:i-1) = 0$, $v(i) = 1$, and $v(i:m)$ is stored on exit in

$A(ia+i-1:ia+m-1, ja+i-1)$; $u(1:i) = 0$, $u(i+1) = 1$, and $u(i+1:n)$ is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$; *tauq* is stored in $TAUQ(ja+i-1)$ and *taup* in $TAUP(ia+i-1)$.

If $m < n$, $v(1:i) = 0$, $v(i+1) = 1$, and $v(i+1:m)$ is stored on exit in

$A(ia+i+1:ia+m-1, ja+i-1)$; $u(1:i-1) = 0$, $u(i) = 1$, and $u(i:n)$ is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$; *tauq* is stored in $TAUQ(ja+i-1)$ and *taup* in $TAUP(ia+i-1)$. The elements of the vectors *v* and *u* together form the *m*-by-*nb* matrix *V* and the *nb*-by-*n* matrix *U'* which are necessary, with *X* and *Y*, to apply the transformation to the unreduced part of the matrix, using a block update of the form: sub(*A*) := sub(*A*) - *V**Y'* - *X**U'*. The contents of sub(*A*) on exit are illustrated by the following examples with *nb* = 2:

$m = 6$ and $n = 5$ ($m > n$):

$$\begin{bmatrix} 1 & 1 & u1 & u1 & u1 \\ v1 & 1 & 1 & u2 & u2 \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \end{bmatrix}$$

$m = 5$ and $n = 6$ ($m < n$):

$$\begin{bmatrix} 1 & u1 & u1 & u1 & u1 & u1 \\ 1 & 1 & u2 & u2 & u2 & u2 \\ v1 & 1 & a & a & a & a \\ v1 & v2 & a & a & a & a \\ v1 & v2 & a & a & a & a \end{bmatrix}$$

where `a` denotes an element of the original matrix which is unchanged, `vi` denotes an element of the vector defining *H*(`i`), and `ui` an element of the vector defining *G*(`i`).

p?lacon

Estimates the 1-norm of a square matrix, using the reverse communication for evaluating matrix-vector products.

Syntax

```
call pslacon (n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
call pdlacon (n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
call pclacon (n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
call pzlacon (n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
```

Description

This routine estimates the 1-norm of a square, real/unitary distributed matrix *A*. Reverse communication is used for evaluating matrix-vector products. `x` and `v` are aligned with the distributed matrix *A*, this information is implicitly contained within *iv*, *ix*, *descv*, and *descx*.

Input Parameters

n (global).`INTEGER`.
The length of the distributed vectors `v` and `x`. $n \geq 0$.

v (local).
`REAL` for `pslacon`
`DOUBLE PRECISION` for `pdlacon`

COMPLEX for pclacon
COMPLEX*16 for pzlacon
Pointer into the local memory to an array of DIMENSION *LOCr*(*n*+mod(*iv*-1, *mb_v*)). On the final return, *v* = *a***w*, where *est* = norm(*v*)/norm(*w*) (*w* is not returned).

iv, *jv* (global) INTEGER. The row and column indices in the global array *v* indicating the first row and the first column of the submatrix *V*, respectively.

descv (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *V*.

x (local).
REAL for pslacon
DOUBLE PRECISION for pdlacon
COMPLEX for pclacon
COMPLEX*16 for pzlacon
Pointer into the local memory to an array of DIMENSION *LOCr*(*n*+mod(*ix*-1, *mb_x*)).

ix, *jx* (global) INTEGER. The row and column indices in the global array *x* indicating the first row and the first column of the submatrix *X*, respectively.

descx (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *X*.

isgn (local).INTEGER.
Array, DIMENSION *LOCr*(*n*+mod(*ix*-1, *mb_x*)). *isgn* is aligned with *x* and *v*.

kase (local).INTEGER.
On the initial call to p?lacon, *kase* should be 0.

Output Parameters

x (local).
On an intermediate return, X should be overwritten by
A **X*, if *kase*=1,
A' **X*, if *kase*=2,

p?lacon must be re-called with all the other parameters unchanged.

est (global).
REAL for single precision flavors
DOUBLE PRECISION for double precision flavors

kase (local) INTEGER.
On an intermediate return, *kase* will be 1 or 2, indicating whether X should be overwritten by $A * X$ or $A' * X$. On the final return from p?lacon, *kase* will again be 0.

p?laconsb

Looks for two consecutive small subdiagonal elements.

```
call pslaconsb (a, desca, i, l, m, h44, h33, h43h34, buf, lwork)
call pdlaconsb (a, desca, i, l, m, h44, h33, h43h34, buf, lwork)
```

Description

This routine looks for two consecutive small subdiagonal elements by seeing the effect of starting a double shift QR iteration given by *h44*, *h33*, and *h43h34* and see if this would make a subdiagonal negligible.

Input Parameters

a (global).
REAL for pslaconsb
DOUBLE PRECISION for pdlaconsb
Array, DIMENSION (*desca* (*lld_*),*). On entry, the Hessenberg matrix whose tridiagonal part is being scanned. Unchanged on exit.

desca (global and local) INTEGER.
Array of DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

i (global) INTEGER.
The global location of the bottom of the unreduced submatrix of A. Unchanged on exit.

l (global) INTEGER.
The global location of the top of the unreduced submatrix of A. Unchanged on exit.

h44,

h33,

h43h34 (global).
`REAL` for `pslaconsb`
`DOUBLE PRECISION` for `pdlaconsb`
These three values are for the double shift *QR* iteration.

lwork (global).`INTEGER`.
This must be at least `7*ceil(ceil((`*i-l*`)/`*hbl* `)/lcm(`*nprow*, *npcol*`))`. Here `lcm` is least common multiple and *nprowxnpcol* is the logical grid size.

Output Parameters

m (global).
On exit, this yields the starting location of the *QR* double shift. This will satisfy: $l \le m \le i$-2.

buf (local).
`REAL` for `pslaconsb`
`DOUBLE PRECISION` for `pdlaconsb`
Array of size *lwork*.

lwork (global).
On exit, *lwork* is the size of the work buffer.

p?lacp2

Copies all or part of a distributed matrix to another distributed matrix.

Syntax

```
call pslacp2 (uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pdlacp2 (uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pclacp2 (uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pzlacp2 (uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
```

Description

This routine copies all or part of a distributed matrix A to another distributed matrix B. No communication is performed, p?lacp2 performs a local copy sub(A) := sub(B), where sub(A) denotes A(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) and sub(B) denotes B(*ib*:*ib*+*m*-1, *jb*:*jb*+*n*-1).

p?lacp2 requires that only dimension of the matrix operands is distributed.

Input Parameters

uplo (global) CHARACTER.
Specifies the part of the distributed matrix sub(A) to be copied:
= 'U': Upper triangular part is copied; the strictly lower triangular part of sub(A) is not referenced;
= 'L': Lower triangular part is copied; the strictly upper triangular part of sub(A) is not referenced.
Otherwise: all of the matrix sub(A) is copied.

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(A). ($m \geq 0$).

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(A). ($n \geq 0$).

a (local).
REAL for pslacp2
DOUBLE PRECISION for pdlacp2
COMPLEX for pclacp2
COMPLEX*16 for pzlacp2.
Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the *m*-by-*n* distributed matrix sub(A).

ia, ja (global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of sub(A), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

ib, jb (global) INTEGER. The row and column indices in the global array B indicating the first row and the first column of sub(B), respectively.

descb (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *B*.

Output Parameters

b (local).
REAL for pslacp2
DOUBLE PRECISION for pdlacp2
COMPLEX for pclacp2
COMPLEX*16 for pzlacp2.
Pointer into the local memory to an array of DIMENSION (*lld_b*, *LOCc*(*jb*+*n*-1)). This array contains on exit the local pieces of the distributed matrix sub(B) set as follows:

if *uplo* = 'U', $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$,
$1 \le i \le j$, $1 \le j \le n$;
if *uplo* = 'L', $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$,
$j \le i \le m$, $1 \le j \le n$;

otherwise, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$,
$1 \le i \le m$, $1 \le j \le n$.

p?lacp3

Copies from a global parallel array into a local replicated array or vice versa.

Syntax

```
call pslacp3 (m, i, a, desca, b, ldb, ii, jj, rev)
call pdlacp3 (m, i, a, desca, b, ldb, ii, jj, rev)
```

Description

This is an auxiliary routine that copies from a global parallel array into a local replicated array or vise versa. Note that the entire submatrix that is copied gets placed on one node or more. The receiving node can be specified precisely, or all nodes can receive, or just one row or column of nodes.

Input Parameters

m (global) INTEGER. *m* is the order of the square submatrix that is copied. $m \geq 0$. Unchanged on exit.

i (global) INTEGER.
A(*i*, *i*) is the global location that the copying starts from. Unchanged on exit.

a (global).
REAL for pslacp3
DOUBLE PRECISION for pdlacp3
Array, DIMENSION (*desca*(*lld_*),*). On entry, the parallel matrix to be copied into or from.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

b (local).
REAL for pslacp3
DOUBLE PRECISION for pdlacp3
Array, DIMENSION (*ldb*, *m*).
If *rev* = 0, this is the global portion of the array *A*(*i*:*i*+*m*-1, *i*:*i*+*m*-1).
If *rev* = 1, this is the unchanged on exit.

ldb (local) INTEGER.
The leading dimension of *B*.

ii (global) INTEGER
By using *rev* 0 and 1, data can be sent out and returned again. If *rev* = 0, then *ii* is destination row index for the node(s) receiving the replicated *B*.
If *ii* $\geq$ 0, *jj* $\geq$ 0, then node (*ii*, *jj*) receives the data.
If *ii* = -1, *jj* $\geq$ 0, then all rows in column *jj* receive the data.
If *ii* $\geq$ 0, *jj* = -1, then all cols in row *ii* receive the data.
f *ii* = -1, *jj* = -1, then all nodes receive the data.
If *rev* !=0, then *ii* is the source row index for the node(s) sending the replicated *B*.

jj (global) INTEGER. Similar description as *ii* above.

rev (global) INTEGER.
Use *rev* = 0 to send global *A* into locally replicated *B* (on node (*ii*, *jj*)).
Use *rev* != 0 to send locally replicated *B* from node (*ii*, *jj*) to its owner (which changes depending on its location in *A*) into the global *A*.

Output Parameters

a (global). On exit, if *rev* = 1, the copied data. Unchanged on exit if *rev* = 0.

b (local). If *rev* = 1, this is unchanged on exit.

p?lacpy

Copies all or part of one two-dimensional array to another.

Syntax

```
call pslacpy (uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pdlacpy (uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pclacpy (uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pzlacpy (uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
```

Description

This routine copies all or part of a distributed matrix *A* to another distributed matrix *B*. No communication is performed, `p?lacpy` performs a local copy sub(*A*) := sub(*B*), where sub(*A*) denotes *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) and sub(*B*) denotes *B*(*ib*:*ib*+*m*-1,*jb*:*jb*+*n*-1).

Input Parameters

uplo (global). `CHARACTER`.
Specifies the part of the distributed matrix sub(*A*) to be copied:
= 'U': Upper triangular part is copied; the strictly lower triangular part of sub(*A*) is not referenced;
= 'L': Lower triangular part is copied; the strictly upper triangular part of sub(*A*) is not referenced.
Otherwise: all of the matrix sub(*A*) is copied.

m (global) `INTEGER`.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). ($m \geq 0$).

n (global) `INTEGER`.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). ($n \geq 0$).

a (local).
REAL for pslacpy
DOUBLE PRECISION for pdlacpy
COMPLEX for pclacpy
COMPLEX*16 for pzlacpy.
Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the distributed matrix sub(*A*).

ia, ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

ib, jb (global) INTEGER. The row and column indices in the global array *B* indicating the first row and the first column of sub(*B*) respectively.

descb (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

b (local).
REAL for pslacpy
DOUBLE PRECISION for pdlacpy
COMPLEX for pclacpy
COMPLEX*16 for pzlacpy.
Pointer into the local memory to an array of DIMENSION (*lld_b*, *LOCc*(*jb*+*n*-1)).
This array contains on exit the local pieces of the distributed matrix sub(B) set as follows:

if *uplo* = 'U', $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$,
$1 \le i \le j$, $1 \le j \le n$;
if *uplo* = 'L', $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$,
$j \le i \le m$, $1 \le j \le n$;

otherwise, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$,
$1 \le i \le m$, $1 \le j \le n$.

p?laevswp

Moves the eigenvectors from where they are computed to ScaLAPACK standard block cyclic array.

Syntax

```
call pslaevswp (n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork,
    lrwork)
call pdlaevswp (n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork,
    lrwork)
call pclaevswp (n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork,
    lrwork)
call pzlaevswp (n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork,
    lrwork)
```

Description

This routine moves the eigenvectors (potentially unsorted) from where they are computed, to a ScaLAPACK standard block cyclic array, sorted so that the corresponding eigenvalues are sorted.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

n (global). `INTEGER`.
The order of the matrix *A*. $n \geq 0$.

zin (local).
`REAL` for `pslaevswp`
`DOUBLE PRECISION` for `pdlaevswp`
`COMPLEX` for `pclaevswp`
`COMPLEX*16` for `pzlaevswp`.
Array, `DIMENSION` (*ldzi*, *nvs*(*iam*)). The eigenvectors on input. Each eigenvector resides entirely in one process. Each process holds a contiguous set of *nvs*(*iam*) eigenvectors. The first eigenvector which the process holds is:
sum for *i*=[0, *iam*-1) of *nvs*(*i*).

ldzi (local) `INTEGER`. The leading dimension of the *zin* array.

iz, jz (global) INTEGER.The row and column indices in the global array *Z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *Z*.

nvs (global) INTEGER.
Array, DIMENSION(*nprocs*+1)
nvs(*i*) = number of processes number of eigenvectors held by processes [0, *i* - 1)
nvs(1) = number of eigen vectors held by [0, 1 - 1) = 0
nvs(*nprocs*+1) = number of eigen vectors held by [0, *nprocs*) = total number of eigenvectors.

key (global) INTEGER.
Array, DIMENSION (*n*). Indicates the actual index (after sorting) for each of the eigenvectors.

rwork (local).
REAL for pslaevswp
DOUBLE PRECISION for pdlaevswp
COMPLEX for pclaevswp
COMPLEX*16 for pzlaevswp.
Array, DIMENSION (*lrwork*).

lrwork (local) INTEGER.
Dimension of *work.*

Output Parameters

z (local).
REAL for pslaevswp
DOUBLE PRECISION for pdlaevswp
COMPLEX for pclaevswp
COMPLEX*16 for pzlaevswp.
Array, global DIMENSION (*n*, *n*), local DIMENSION (*descz*(*dlen_*), *nq*). The eigenvectors on output. The eigenvectors are distributed in a block cyclic manner in both dimensions, with a block size of *nb*.

p?lahrd

Reduces the first nb *columns of a general rectangular matrix A so that elements below the k^th^ subdiagonal are zero, by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A.*

Syntax

```
call pslahrd (n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
call pdlahrd (n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
call pclahrd (n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
call pzlahrd (n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
```

Description

The routines reduces the first *nb* columns of a real general *n*-by-(n-k+1) distributed matrix A(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-*k*) so that elements below the *k*-th subdiagonal are zero. The reduction is performed by an orthogonal/unitary similarity transformation $Q' * A * Q$. The routine returns the matrices V and T which determine Q as a block reflector $I - V*T*V'$, and also the matrix $Y = A * V * T$.

This is an auxiliary routine called by p?gehrd. In the following comments sub(A) denotes A(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1).

Input Parameters

n (global) INTEGER. The order of the distributed submatrix sub(A). $n \ge 0$.

k (global) INTEGER. The offset for the reduction. Elements below the *k*-th subdiagonal in the first *nb* columns are reduced to zero.

nb (global) INTEGER. The number of columns to be reduced.

a (local).
REAL for pslahrd
DOUBLE PRECISION for pdlahrd
COMPLEX for pclahrd
COMPLEX*16 for pzlahrd.

Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-*k*)). On entry, this array contains the the local pieces of the *n*-by-(*n*-*k*+1) general distributed matrix *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-*k*).

ia, *ja* (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

iy, *jy* (global) INTEGER. The row and column indices in the global array *Y* indicating the first row and the first column of the submatrix sub(*Y*), respectively.

descy (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *Y*.

work (local).
REAL for pslahrd
DOUBLE PRECISION for pdlahrd
COMPLEX for pclahrd
COMPLEX*16 for pzlahrd.
Array, DIMENSION (*nb*).

Output Parameters

a (local).
On exit, the elements on and above the *k*-th subdiagonal in the first *nb* columns are overwritten with the corresponding elements of the reduced distributed matrix; the elements below the *k*-th subdiagonal, with the array *tau*, represent the matrix *Q* as a product of elementary reflectors. The other columns of *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-*k*) are unchanged. See *Application Notes* below.

tau (local)

REAL for pslahrd
DOUBLE PRECISION for pdlahrd
COMPLEX for pclahrd
COMPLEX*16 for pzlahrd.
Array, DIMENSION *LOCc*(*ja*+*n*-2).
The scalar factors of the elementary reflectors (see *Application Notes* below). *tau* is tied to the distributed matrix *A*.

t (local)
REAL for pslahrd
DOUBLE PRECISION for pdlahrd

COMPLEX for pclahrd
COMPLEX*16 for pzlahrd.
Array, DIMENSION (*nb_a*, *nb_a*)
The upper triangular matrix *T*.

y (local).
REAL for pslahrd
DOUBLE PRECISION for pdlahrd
COMPLEX for pclahrd
COMPLEX*16 for pzlahrd.
Pointer into the local memory to an array of DIMENSION (*lld_y*, *nb_a*). On exit, this array contains the local pieces of the *n*-by-*nb* distributed matrix *Y*.
$lld\_y \geq LOCr(ia+n-1)$.

Application Notes

The matrix *Q* is represented as a product of *nb* elementary reflectors

$Q = H(1)\ H(2)\ .\ .\ .\ H(nb)$.

Each *H*(*i*) has the form

$H(i) = i - tau * v * v'$,

where *tau* is a real/complex scalar, and *v* is a real/complex vector with $v(1:i+k-1) = 0$, $v(i+k) = 1$; $v(i+k+1:n)$ is stored on exit in $A(ia+i+k:ia+n-1, ja+i-1)$, and *tau* in $TAU(ja+i-1)$.

The elements of the vectors *v* together form the $(n-k+1)$-by-*nb* matrix *V* which is needed, with *T* and *Y*, to apply the transformation to the unreduced part of the matrix, using an update of the form: $A(ia:ia+n-1, ja:ja+n-k) := (I-V*T*V')*(A(ia:ia+n-1, ja:ja+n-k)-Y*V')$. The contents of $A(ia:ia+n-1, ja:ja+n-k)$ on exit are illustrated by the following example with $n = 7$, $k = 3$, and $nb = 2$:

$$\begin{bmatrix} a & h & a & a & a \\ a & h & a & a & a \\ a & h & a & a & a \\ h & h & a & a & a \\ v1 & h & a & a & a \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \end{bmatrix}$$

where `a` denotes an element of the original matrix A(`ia`:`ia`+`n`-1, `ja`:`ja`+`n`-`k`), `h` denotes a modified element of the upper Hessenberg matrix H, and `vi` denotes an element of the vector defining H(`i`).

p?laiect

Exploits IEEE arithmetic to accelerate the computations of eigenvalues. (C interface function).

Syntax

```
void pslaiect (float *sigma, int *n, float *d, int *count);
void pdlaiectb (float *sigma, int *n, float *d, int *count);
void pdlaiectl (float *sigma, int *n, float *d, int *count);
```

Description

This routine computes the number of negative eigenvalues of (A- σI). This implementation of the Sturm Sequence loop exploits IEEE arithmetic and has no conditionals in the innermost loop. The signbit for real routine `pslaiect` is assumed to be bit 32. Double precision routines `pdlaiectb` and `pdlaiectl` differ in the order of the double precision word storage and, consequently, in the signbit location. For `pdlaiectb`, the double precision word is stored in the big-endian word order and the signbit is assumed to be bit 32. For `pdlaiectl`, the double precision word is stored in the little-endian word order and the signbit is assumed to be bit 64.

Note that all arguments are call-by-reference so that this routine can be directly called from Fortran code.

This is a ScaLAPACK internal subroutine and arguments are not checked for unreasonable values.

Input Parameters

sigma — `REAL` for `pslaiect`
`DOUBLE PRECISION` for `pdlaiectb/pdlaiectl`.
The shift. `p?laiect` finds the number of eigenvalues less than equal to *sigma*.

n — `INTEGER`.
The order of the tridiagonal matrix T. $n \geq 1$.

d REAL for pslaiect
DOUBLE PRECISION for pdlaiectb/pdlaiectl.
Array of DIMENSION $(2n - 1)$.
On entry, this array contains the diagonals and the squares of the off-diagonal elements of the tridiagonal matrix *T*. These elements are assumed to be interleaved in memory for better cache performance. The diagonal entries of *T* are in the entries $d(1)$, $d(3)$, ..., $d(2n-1)$, while the squares of the off-diagonal entries are $d(2)$, $d(4)$, ..., $d(2n-2)$. To avoid overflow, the matrix must be scaled so that its largest entry is no greater than $overflow^{(1/2)} * underflow^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.

Output Parameters

n INTEGER.
The count of the number of eigenvalues of *T* less than or equal to *sigma*.

p?lange

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a general rectangular matrix.

Syntax

```
val = pslange (norm, m, n, a, ia, ja, desca, work)
val = pdlange (norm, m, n, a, ia, ja, desca, work)
val = pclange (norm, m, n, a, ia, ja, desca, work)
val = pzlange (norm, m, n, a, ia, ja, desca, work)
```

Description

The function returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a distributed matrix sub(*A*) = *A*(ia:ia+m-1, ja:ja+n-1).

p?lange returns the value

(max(abs($A(i,j)$)), *norm* = 'M' or 'm' with $ia \le i \le ia+m-1$,
(and $ja \le j \le ja+n-1$,

(

(*norm1*(sub(A)), *norm* = '1', 'O' or 'o'

(

(*normI*(sub(A)), *norm* = 'I' or 'i'

(

(*normF*(sub(A)), *norm* = 'F', 'f', 'E' or 'e',

where *norm1* denotes the 1-norm of a matrix (maximum column sum), *normI* denotes the infinity norm of a matrix (maximum row sum) and *normF* denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs($A(i,j)$)) is not a matrix norm.

Input Parameters

norm (global) CHARACTER.
Specifies the value to be returned in p?lange as described above.

m (global). INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(A). When $m = 0$, p?lange is set to zero. $m \geq 0$.

n (global). INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(A). When $n = 0$, p?lange is set to zero. $n \geq 0$.

a (local).
REAL for pslange
DOUBLE PRECISION for pdlange
COMPLEX for pclange
COMPLEX*16 for pzlange.
Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1)) containing the local pieces of the distributed matrix sub(A).

ia, ja (global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix sub(A), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

work (local).
REAL for pslange
DOUBLE PRECISION for pdlange
COMPLEX for pclange
COMPLEX*16 for pzlange.

Array `DIMENSION` (*lwork*).

lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),

*nq*0 if *norm* = '1', 'O' or 'o',

*mp*0 if *norm* = 'I' or 'i',

0 if *norm* = 'F', 'f', 'E' or 'e' (not referenced),

where

iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia, mb_a, myrow, rsrc_a, nprow*),

iacol = indxg2p(*ja, nb_a, mycol, csrc_a, npcol*),

*mp*0 = numroc(*m+iroffa, mb_a, myrow, iarow, nprow*),

*nq*0 = numroc(*n+icoffa, nb_a, mycol, iacol, npcol*),

`indxg2p` and `numroc` are ScaLAPACK tool functions; *myrow, mycol, nprow,* and *npcol* can be determined by calling the subroutine `blacs_gridinfo`.

Output Parameters

val The value returned by the fuction.

p?lanhs

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of an upper Hessenberg matrix.

Syntax

```
val = pslanhs (norm, n, a, ia, ja, desca, work)
val = pdlanhs (norm, n, a, ia, ja, desca, work)
val = pclanhs (norm, n, a, ia, ja, desca, work)
val = pzlanhs (norm, n, a, ia, ja, desca, work)
```

Description

The function returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a distributed matrix sub(*A*) = *A*(*ia:ia+m-1, ja:ja+n-1*).

`p?lanhs` returns the value

(max(abs($A(i,j)$)), *norm* = 'M' or 'm' with $ia \le i \le ia+m-1$,
(and $ja \le j \le ja+n-1$,

(

(*norm1*(sub(A)), *norm* = '1', 'O' or 'o'

(

(*normI*(sub(A)), *norm* = 'I' or 'i'

(

(*normF*(sub(A)), *norm* = 'F', 'f', 'E' or 'e',

where *norm1* denotes the 1-norm of a matrix (maximum column sum), *normI* denotes the infinity norm of a matrix (maximum row sum) and *normF* denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs($A(i,j)$)) is not a matrix norm.

Input Parameters

norm (global) CHARACTER.
Specifies the value to be returned in p?lange as described above.

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(A). When $n = 0$, p?lanhs is set to zero. $n \ge 0$.

a (local).
REAL for pslanhs
DOUBLE PRECISION for pdlanhs
COMPLEX for pclanhs
COMPLEX*16 for pzlanhs
Pointer into the local memory to an array of DIMENSION (*lld_a*, $LOCc(ja+n-1)$) containing the local pieces of the distributed matrix sub(A).

ia,ja (global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix sub(A), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

work (local).
REAL for pslanhs
DOUBLE PRECISION for pdlanhs
COMPLEX for pclanhs

`COMPLEX*16` for `pzlanh`.
Array, `DIMENSION` (*lwork*).
lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),
*nq*0 if *norm* = '1', 'O' or 'o',
*mp*0 if *norm* = 'I' or 'i',
0 if *norm* = 'F', 'f', 'E' or 'e' (not referenced),
where
iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia, mb_a, myrow, rsrc_a, nprow*),
iacol = indxg2p(*ja, nb_a, mycol, csrc_a, npcol*),
*mp*0 = numroc(*m+iroffa, mb_a, myrow, iarow, nprow*),
*nq*0 = numroc(*n+icoffa, nb_a, mycol, iacol, npcol*),
`indxg2p` and `numroc` are ScaLAPACK tool functions; *myrow, mycol, nprow*, and *npcol* can be determined by calling the subroutine `blacs_gridinfo`.

Output Parameters

val The value returned by the fuction.

p?lansy, p?lanhe

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a real symmetric or a complex Hermitian matrix.

Syntax

```
val = pslansy (norm, uplo, n, a, ia, ja, desca, work)
val = pdlansy (norm, uplo, n, a, ia, ja, desca, work)
val = pclansy (norm, uplo, n, a, ia, ja, desca, work)
val = pzlansy (norm, uplo, n, a, ia, ja, desca, work)
val = pclanhe (norm, uplo, n, a, ia, ja, desca, work)
val = pzlanhe (norm, uplo, n, a, ia, ja, desca, work)
```

Description

The functions return the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a distributed matrix sub(A) = A(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1).

`p?lansy, p?lanhe` return the value

(max(abs(A(*i*,*j*))), *norm* = 'M' or 'm' with *ia* ≤ *i* ≤ *ia*+*m*-1,
(and *ja* ≤ *j* ≤ *ja*+*n*-1,

(

(*norm1*(sub(A)), *norm* = '1', 'O' or 'o'

(

(*normI*(sub(A)), *norm* = 'I' or 'i'

(

(*normF*(sub(A)), *norm* = 'F', 'f', 'E' or 'e',

where *norm1* denotes the 1-norm of a matrix (maximum column sum), *normI* denotes the infinity norm of a matrix (maximum row sum) and *normF* denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A(*i*,*j*))) is not a matrix norm.

Input Parameters

norm (global) `CHARACTER`.
Specifies the value to be returned in `p?lange` as described above.

uplo (global) `CHARACTER`.
Specifies whether the upper or lower triangular part of the symmetric matrix sub(A) is to be referenced.

= 'U': Upper triangular part of sub(A) is referenced,

= 'L': Lower triangular part of sub(A) is referenced.

n (global) `INTEGER`.
The number of columns to be operated on i.e the number of columns of the distributed submatrix sub(A). When $n = 0$, `p?lansy` is set to zero. $n \geq 0$.

a (local).
`REAL` for `pslansy`
`DOUBLE PRECISION` for `pdlansy`
`COMPLEX` for `pclansy, pclanhe`

COMPLEX*16 for pzlansy, pzlanhe.
Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1)) containing the local pieces of the distributed matrix sub(*A*).
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular matrix which norm is to be computed, and the strictly lower triangular part of this matrix is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular matrix which norm is to be computed, and the strictly upper triangular part of sub(*A*) is not referenced.

ia,ja (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local).
REAL for pslansy
DOUBLE PRECISION for pdlansy
COMPLEX for pclansy, pclanhe
COMPLEX*16 for pzlansy, pzlanhe.
Array DIMENSION (*lwork*).
lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),
2**nq*0+*np*0+*ldw* if *norm* = '1', 'O' or 'o', 'I' or 'i',

where *ldw* is given by:

```
if( nprow.ne.npcol ) then
   ldw = mb_a*ceil(ceil(np0/mb_a)/(lcm/nprow))
else
   ldw = 0
end if
```

0 if *norm* = 'F', 'f', 'E' or 'e' (not referenced),

where *lcm* is the least common multiple of *nprow* and *npcol*
lcm = ilcm(*nprow, npcol*) and ceil denotes the ceiling operation (iceil).
iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia, mb_a, myrow, rsrc_a, nprow*),
iacol = indxg2p(*ja, nb_a, mycol, csrc_a, npcol*),
*mp*0 = numroc(*m*+*iroffa, mb_a, myrow, iarow, nprow*),
*nq*0 = numroc(*n*+*icoffa, nb_a, mycol, iacol, npcol*),
indxg2p and numroc are ScaLAPACK tool functions; *myrow, mycol, nprow,* and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

val The value returned by the fuction.

p?lantr

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a triangular matrix.

Syntax

```
val = pslantr (norm, uplo, diag, m, n, a, ia, ja, desca, work)
val = pdlantr (norm, uplo, diag, m, n, a, ia, ja, desca, work)
val = pclantr (norm, uplo, diag, m, n, a, ia, ja, desca, work)
val = pzlantr (norm, uplo, diag, m, n, a, ia, ja, desca, work)
```

Description

The function returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular distributed matrix sub(A) = A(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1).

`p?lantr` returns the value

(max(abs(A(*i*,*j*))), *norm* = 'M' or 'm' with *ia* $\leq$ *i* $\leq$ *ia*+*m*-1,
(and *ja* $\leq$ *j* $\leq$ *ja*+*n*-1,

(

(*norm1*(sub(A)), *norm* = '1', 'O' or 'o'

(

(*normI*(sub(A)), *norm* = 'I' or 'i'

(

(*normF*(sub(A)), *norm* = 'F', 'f', 'E' or 'e', where *norm1* denotes the 1-norm of a matrix (maximum column sum), *normI* denotes the infinity norm of a matrix (maximum row sum) and *normF* denotes the Frobenius norm of a matrix (square root of sum of squares). Note that `max(abs(`*A*`(`*i,j*`)))` is not a matrix norm.

Input Parameters

norm (global) `CHARACTER.`
Specifies the value to be returned in `p?lantr` as described above.

uplo (global) `CHARACTER.`
Specifies whether the upper or lower triangular part of the symmetric matrix sub(*A*) is to be referenced.

= 'U': Upper trapezoidal,
= 'L': Lower trapezoidal.

Note that sub(*A*) is triangular instead of trapezoidal if *m* = *n*.

diag (global). `CHARACTER.`
Specifies whether or not the distributed matrix sub(*A*) has unit diagonal.
= 'N': Non-unit diagonal.
= 'U': Unit diagonal.

m (global) `INTEGER.`
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(A). When $m = 0$, `p?lantr` is set to zero. $m \geq 0$.

n (global) `INTEGER.`
The number of columns to be operated on i.e the number of columns of the distributed submatrix sub(*A*). When $n = 0$, `p?lantr` is set to zero. $n \geq 0$.

a (local).
`REAL` for `pslantr`
`DOUBLE PRECISION` for `pdlantr`
`COMPLEX` for `pclantr`
`COMPLEX*16` for `pzlantr.`
Pointer into the local memory to an array of `DIMENSION` (*lld_a*, *LOCc*(*ja*+*n*-1)) containing the local pieces of the distributed matrix sub(*A*).

ia,ja (global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) `INTEGER` array, `DIMENSION` (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local).
REAL for pslantr
DOUBLE PRECISION for pdlantr
COMPLEX for pclantr
COMPLEX*16 for pzlantr.
Array DIMENSION (*lwork*).
lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),
nq0 if *norm* = '1', 'O' or 'o',
mp0 if *norm* = 'I' or 'i',
0 if *norm* = 'F', 'f', 'E' or 'e' (not referenced),

where *lcm* is the least common multiple of *nprow* and *npcol*
lcm = ilcm(*nprow*, *npcol*) and ceil denotes the ceiling operation (iceil).
iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia, mb_a, myrow, rsrc_a, nprow*),
iacol = indxg2p(*ja, nb_a, mycol, csrc_a, npcol*),
mp0 = numroc(*m+iroffa, mb_a, myrow, iarow, nprow*),
nq0 = numroc(*n+icoffa, nb_a, mycol, iacol, npcol*),
indxg2p and numroc are ScaLAPACK tool functions; *myrow, mycol, nprow,* and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

val The value returned by the fuction.

p?lapiv

Applies a permutation matrix to a general distributed matrix, resulting in row or column pivoting.

Syntax

```
call pslapiv (direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp,
    descip, iwork)
call pdlapiv  (direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp,
    descip, iwork)
call pclapiv  (direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp,
    descip, iwork)
```

```
call pzlapiv (direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp,
    descip, iwork)
```

Description

This routine applies either *P* (permutation matrix indicated by *ipiv*) or inv(*P*) to a general *m*-by-*n* distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1), resulting in row or column pivoting. The pivot vector may be distributed across a process row or a column. The pivot vector should be aligned with the distributed matrix *A*. This routine will transpose the pivot vector, if necessary.

For example, if the row pivots should be applied to the columns of sub(*A*), pass *rowcol*='C' and *pivroc*='C'.

Input Parameters

direc (global) CHARACTER*1.

Specifies in which order the permutation is applied:
= 'F' (Forward). Applies pivots Forward from top of matrix.
Computes *P**sub(*A*).
= 'B' (Backward) Applies pivots Backward from bottom of matrix.
Computes inv(*P*)*sub(*A*).

rowcol (global) CHARACTER*1.
Specifies if the rows or columns are to be permuted:

= 'R' Rows will be permuted,
= 'C' Columns will be permuted.

pivroc (global) CHARACTER*1.
Specifies whether *ipiv* is distributed over a process row or column:
= 'R' *ipiv* is distributed over a process row,
= 'C' *ipiv* is distributed over a process column.

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). When $m = 0$, p?lapiv is set to zero. $m \geq 0$.

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). When $n = 0$, p?lapiv is set to zero. $n \geq 0$.

a (local).
REAL for pslapiv
DOUBLE PRECISION for pdlapiv
COMPLEX for pclapiv

COMPLEX*16 for pzlapiv.
Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1)) containing the local pieces of the distributed matrix sub(*A*).

ia,ja (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

ipiv (local).INTEGER.
Array, DIMENSION (*lipiv*) where *lipiv* is when *rowcol*='R' or 'r':

≥ *LOCr*(*ia*+*m*-1) + *mb_a* if *pivroc*='C' or 'c',
≥ *LOCc*(*m* + mod(*jp*-1, *nb_p*)) if *pivroc*='R' or 'r', and,

when *rowcol*='C' or 'c':

≥ *LOCr* (*n* + mod(*ip*-1, *mb_p*)) if *pivroc*='C' or 'c',
≥ *LOCc* (*ja*+*n*-1) + *nb_a* if *pivroc*='R' or 'r'.

This array contains the pivoting information. *ipiv(i)* is the global row (column), local row (column) *i* was swapped with. When *rowcol*='R' or 'r' and *pivroc*='C' or 'c', or *rowcol*='C' or 'c' and *pivroc*='R' or 'r', the last piece of this array of size *mb_a* (resp. *nb_a*) is used as workspace. In those cases, this array is tied to the distributed matrix *A*.

ip,jp (global) INTEGER. The row and column indices in the global array *P* indicating the first row and the first column of the submatrix sub(*P*), respectively.

descip (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed vector *ipiv*.

iwork (local). INTEGER.
Array, DIMENSION (*ldw*), where *ldw* is equal to the workspace necessary for transposition, and the storage of the tranposed *ipiv* :

Let *lcm* be the least common multiple of *nprow* and *npcol*.

```
If( rowcol.eq.'r' .and. pivroc.eq.'r' ) then
   If( nprow.eq.npcol ) then
      ldw = LOCr( n_p + mod(jp-1, nb_p) ) + nb_p
   else
      ldw = LOCr( n_p + mod(jp-1, nb_p) ) +
            nb_p * ceil( ceil(LOCc(n_p)/nb_p) / (lcm/npcol) )
   end if
else if( rowcol.eq.'c' .and. pivroc.eq.'c' ) then
```

```
          if( nprow.eq.npcol ) then
              ldw = LOCc( m_p + mod(ip-1, mb_p) ) + mb_p
          else
               ldw = LOCc( m_p + mod(ip-1, mb_p) ) +
                      mb_p * ceil(ceil(LOCr(m_p)/mb_p) / (lcm/nprow) )
          end if
     else
          iwork is not referenced.
     end if.
```

Output Parameters

a (local).
On exit, the local pieces of the permuted distributed submatrix.

p?laqge

Scales a general rectangular matrix, using row and column scaling factors computed by `p?geequ` .

Syntax

```
call pslaqge (m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
call pdlaqge (m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
call pclaqge (m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
call pzlaqge (m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
```

Description

This routine equilibrates a general *m*-by-*n* distributed matrix sub(A) = A(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) using the row and scaling factors in the vectors *r* and *c* computed by p?geequ.

Input Parameters

m (global). `INTEGER`.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(A). ($m \geq 0$).

n (global).INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). ($n \geq 0$).

a (local).
REAL for pslaqge
DOUBLE PRECISION for pdlaqge
COMPLEX for pclaqge
COMPLEX*16COMPLEX*16 for pzlaqge.
Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the distributed matrix sub(*A*).

ia,ja (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

r (local).
REAL for pslaqge
DOUBLE PRECISION for pdlaqge
COMPLEX for pclaqge
COMPLEX*16 for pzlaqge.
Array, DIMENSION *LOCr*(*m_a*). The row scale factors for sub(*A*). *r* is aligned with the distributed matrix *A*, and replicated across every process column. *r* is tied to the distributed matrix *A*.

c (local).
REAL for pslaqge
DOUBLE PRECISION for pdlaqge
COMPLEX for pclaqge
COMPLEX*16 for pzlaqge.
Array, DIMENSION *LOCc*(*n_a*). The row scale factors for sub(*A*). *c* is aligned with the distributed matrix *A*, and replicated across every process column. *c* is tied to the distributed matrix *A*.

rowcnd (local).
REAL for pslaqge
DOUBLE PRECISION for pdlaqge
COMPLEX for pclaqge
COMPLEX*16 for pzlaqge.
The global ratio of the smallest *r(i)* to the largest *r(i)*, $ia \leq i \leq ia+m-1$.

colcnd (local).
REAL for pslaqge
DOUBLE PRECISION for pdlaqge
COMPLEX for pclaqge
COMPLEX*16 for pzlaqge.
The global ratio of the smallest *c(i)* to the largest *r(i)*, $ia \le i \le ia+n-1$.

amax (global).
REAL for pslaqge
DOUBLE PRECISION for pdlaqge
COMPLEX for pclaqge
COMPLEX*16 for pzlaqge.
Absolute value of largest distributed submatrix entry.

Output Parameters

a (local).
On exit, the equilibrated distributed matrix. See *equed* for the form of the equilibrated distributed submatrix.

equed (global) CHARACTER.

Specifies the form of equilibration that was done.

= 'N': No equilibration
= 'R': Row equilibration, that is, sub(*A*) has been pre-multiplied by diag(*r*(*ia*:*ia*+*m*-1)),
= 'C': Column equilibration, that is, sub(*A*) has been post-multiplied by diag(*c*(*ja*:*ja*+*n*-1)),
= 'B': Both row and column equilibration, that is, sub(*A*) has been replaced by diag(*r*(*ia*:*ia*+*m*-1)) * sub(*A*) * diag(*c*(*ja*:*ja*+*n*-1)).

p?laqsy

Scales a symmetric/Hermitian matrix, using scaling factors computed by p?poequ .

Syntax

```
call pslaqsy (uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
call pdlaqsy (uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
```

```
call pclaqsy (uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
call pzlaqsy (uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
```

Description

This routine equilibrates a symmetric distributed matrix sub(A) = A(`ia`:`ia`+`n`-1, `ja`:`ja`+`n`-1) using the scaling factors in the vectors `sr` and `sc`. The scaling factors are computed by p?poequ .

Input Parameters

uplo (global) `CHARACTER`.
Specifies the upper or lower triangular part of the symmetric distributed matrix sub(A)is to be referenced:
= 'U': Upper triangular part;
= 'L': Lower triangular part.

n (global) `INTEGER`. The order of the distributed submatrix sub(A). $n \geq 0$.

a (local).
`REAL` for `pslaqsy`
`DOUBLE PRECISION` for `pdlaqsy`
`COMPLEX` for `pclaqsy`
`COMPLEX*16` for `pzlaqsy`.
Pointer into the local memory to an array of `DIMENSION` (*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the distributed matrix sub(A). On entry, the local pieces of the distributed symmetric matrix sub(A).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(A) contains the upper triangular part of the matrix, and the strictly lower triangular part of sub(A) is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(A) contains the lower triangular part of the matrix, and the strictly upper triangular part of sub(A) is not referenced.

ia, ja (global) `INTEGER`. The row and column indices in the global array A indicating the first row and the first column of the submatrix sub(A), respectively.

desca (global and local) `INTEGER` array, `DIMENSION` (*dlen_*). The array descriptor for the distributed matrix A.

sr (local)
`REAL` for `pslaqsy`
`DOUBLE PRECISION` for `pdlaqsy`
`COMPLEX` for `pclaqsy`

COMPLEX*16 for pzlaqsy.
Array, DIMENSION *LOCr*(*m_a*). The scale factors for *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1). *sr* is aligned with the distributed matrix *A*, and replicated across every process column. *sr* is tied to the distributed matrix *A*.

sc (local)
REAL for pslaqsy
DOUBLE PRECISION for pdlaqsy
COMPLEX for pclaqsy
COMPLEX*16 for pzlaqsy.
Array, DIMENSION *LOCc*(*m_a*). The scale factors for *A* (*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1). *sr* is aligned with the distributed matrix *A*, and replicated across every process column. *sr* is tied to the distributed matrix *A*.

scond (global).
REAL for pslaqsy
DOUBLE PRECISION for pdlaqsy
COMPLEX for pclaqsy
COMPLEX*16 for pzlaqsy.
Ratio of the smallest *sr(i)* (respectively *sc(j)*) to the largest *sr(i)* (respectively *sc(j)*), with $ia \le i \le ia+n-1$ and $ja \le j \le ja+n-1$.

amax (global).
REAL for pslaqsy
DOUBLE PRECISION for pdlaqsy
COMPLEX for pclaqsy
COMPLEX*16 for pzlaqsy.
Absolute value of largest distributed submatrix entry.

Output Parameters

a On exit, if *equed* = 'Y', the equilibrated matrix:
diag(*sr*(*ia*:*ia*+*n*-1)) * sub(*A*) * diag(*sc*(*ja*:*ja*+*n*-1)).

equed (global) CHARACTER*1.

Specifies whether or not equilibration was done.
= 'N': No equilibration.
= 'Y': Equilibration was done, that is, sub(*A*) has been replaced by:
diag(*sr*(*ia*:*ia*+*n*-1)) * sub(*A*) * diag(*sc*(*ja*:*ja*+*n*-1)).

p?lared1d

Redistributes an array assuming that the input array, `bycol`, is distributed across rows and that all process columns contain the same copy of `bycol`.

Syntax

```
call pslared1d (n, ia, ja, desc, bycol, byall, work, lwork)
call pdlared1d (n, ia, ja, desc, bycol, byall, work, lwork)
```

Description

This routine redistributes a 1D array. It assumes that the input array *bycol* is distributed across rows and that all process column contain the same copy of *bycol*. The output array *byall* is identical on all processes and contains the entire array.

Input Parameters

np = Number of local rows in *bycol*()

n — (global). INTEGER.
The number of elements to be redistributed. $n \geq 0$.

ia, ja — (global) INTEGER. *ia*, *ja* must be equal to 1.

desc — (global and local) INTEGER array, DIMENSION 8. A 2d array descirptor, which describes *bycol*.

bycol — (local).
REAL for pslared1d
DOUBLE PRECISION for pdlared1d
COMPLEX for pclared1d
COMPLEX*16 for pzlared1d.
Distributed block cyclic array global DIMENSION (*n*), local DIMENSION *np*. *bycol* is distributed across the process rows. All process columns are assumed to contain the same value.

work — (local).
REAL for pslared1d
DOUBLE PRECISION for pdlared1d

COMPLEX for pclared1d
COMPLEX*16 for pzlared1d.
DIMENSION (*lwork*). Used to hold the buffers sent from one process to another.

lwork (local) INTEGER.
The size of the *work* array. *lwork* ≥ numroc(*n*, *desc*(*nb_*), *0*, *0*, *npcol*).

Output Parameters

byall (global).
REAL for pslared1d
DOUBLE PRECISION for pdlared1d
COMPLEX for pclared1d
COMPLEX*16 for pzlared1d.
Global DIMENSION(*n*), local DIMENSION (*n*). *byall* is exactly duplicated on all processes. It contains the same values as *bycol*, but it is replicated across all processes rather than being distributed.

p?lared2d

Redistributes an array assuming that the input array byrow *is distributed across columns and that all process rows contain the same copy of* byrow*.*

Syntax

```
call pslared2d (n, ia, ja, desc, byrow, byall, work, lwork)
call pdlared2d (n, ia, ja, desc, byrow, byall, work, lwork)
```

Description

This routine redistributes a 1D array.

It assumes that the input array *byrow* is distributed across columns and that all process rows contain the same copy of *byrow*. The output array *byall* will be identical on all processes and will contain the entire array.

Input Parameters

np = Number of local rows in *byrow*()

n (global) INTEGER.
The number of elements to be redistributed. $n \geq 0$.

ia, ja (global) INTEGER. *ia*, *ja* must be equal to 1.

desc (global and local) INTEGER array, DIMENSION *(dlen_)*. A 2d array descirptor, which describes *byrow*.

byrow (local).
REAL for pslared2d
DOUBLE PRECISION for pdlared2d
COMPLEX for pclared2d
COMPLEX*16 for pzlared2d.
Distributed block cyclic array global DIMENSION (*n*), local DIMENSION *np*. *bycol* is distributed across the process columns. All process rows are assumed to contain the same value.

work (local).
REAL for pslared2d
DOUBLE PRECISION for pdlared2d
COMPLEX for pclared2d
COMPLEX*16 for pzlared2d.
DIMENSION (*lwork*). Used to hold the buffers sent from one process to another.

lwork (local).INTEGER.
The size of the *work* array. *lwork* ≥ numroc(*n*, *desc*(*nb_*), *0*, *0*, *npcol*).

Output Parameters

byall (global).
REAL for pslared2d
DOUBLE PRECISION for pdlared2d
COMPLEX for pclared2d
COMPLEX*16 for pzlared2d.
Global DIMENSION(*n*), local DIMENSION (*n*). *byall* is exactly duplicated on all processes. It contains the same values as *bycol*, but it is replicated across all processes rather than being distributed.

p?larf

Applies an elementary reflector to a general rectangular matrix.

Syntax

```
call pslarf (side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pdlarf (side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pclarf (side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pzlarf (side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
```

Description

This routine applies a real/complex elementary reflector Q (or Q^T) to a real/complex *m*-by-*n* distributed matrix sub(C) = C(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1), from either the left or the right. Q is represented in the form

$Q = I$ - *tau* * *v* * *v'*,

where *tau* is a real/complex scalar and *v* is a real/complex vector.

If *tau* = 0, then Q is taken to be the unit matrix.

Input Parameters

side (global). CHARACTER.
= 'L': form Q * sub(C),
= 'R': form sub(C) * Q, $Q = Q^T$.

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(A). ($m \geq 0$).

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(A). ($n \geq 0$).

v (local).
REAL for pslarf
DOUBLE PRECISION for pdlarf
COMPLEX for pclarf

COMPLEX*16 for pzlarf.
Pointer into the local memory to an array of DIMENSION (*lld_v*,*) containing the local pieces of the distributed vectors *V* representing the Householder transformation *Q*,

v(*iv*:*iv*+*m*-1, *jv*) if *side* = 'L' and *incv* = 1,
v(*iv*, *jv*:*jv*+*m*-1) if *side* = 'L' and *incv* = *m_v*,
v(*iv*:*iv*+*n*-1, *jv*) if *side* = 'R' and *incv* = 1,
v(*iv*, *jv*:*jv*+*n*-1) if *side* = 'R' and *incv* = *m_v*.

The vector *v* is the representation of *Q*. *v* is not used if *tau* = 0.

iv,jv (global) INTEGER. The row and column indices in the global array *V* indicating the first row and the first column of the submatrix sub(*V*), respectively.

descv (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *V*.

incv (global) INTEGER. The global increment for the elements of *v*. Only two values of *incv* are supported in this version, namely 1 and *m_v*.
incv must not be zero.

tau (local).
REAL for pslarf
DOUBLE PRECISION for pdlarf
COMPLEX for pclarf
COMPLEX*16 for pzlarf.
Array, DIMENSION *LOCc*(*jv*) if *incv* = 1, and *LOCr*(*iv*) otherwise. This array contains the Householder scalars related to the Householder vectors.
tau is tied to the distributed matrix *v*.

c (local).
REAL for pslarf
DOUBLE PRECISION for pdlarf
COMPLEX for pclarf
COMPLEX*16 for pzlarf.
Pointer into the local memory to an array of DIMENSION (*lld_c*, *LOCc*(*jc*+*n*-1)), containing the local pieces of sub(*C*).

ic,jc (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix sub(*C*), respectively.

descc (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local).

REAL for pslarf
DOUBLE PRECISION for pdlarf
COMPLEX for pclarf
COMPLEX*16 for pzlarf.
Array, DIMENSION (*lwork*).

```
If incv = 1,
  if side = 'L',
    if ivcol = iccol,
      lwork ≥ nqc0
    else
      lwork ≥ mpc0 + max( 1, nqc0 )
    end if
  else if side = 'R',
  lwork≥nqc0+max(max(1,mpc0),numroc(numroc(n+
icoffc,nb_v,0,0,npcol),nb_v,0,0,lcmq ) )
  end if
else if incv = m_v,
  if side = 'L',
    lwork ≥ mpc0 + max( max( 1, nqc0 ), numroc(
numroc(m+iroffc,mb_v,0,0,nprow ),mb_v,0,0, lcmp ) )
  else if side = 'R',
    if ivrow = icrow,
      lwork ≥ mpc0
    else
      lwork ≥ nqc0 + max( 1, mpc0 )
    end if
  end if
end if,
```

where *lcm* is the least common multiple of *nprow* and *npcol* and *lcm* = *ilcm*(*nprow*, *npcol*), *lcmp* = *lcm* / *nprow*, *lcmq* = *lcm* / *npcol*,

iroffc = mod(*ic*-1, *mb_c*), *icoffc* = *mod*(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),
iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),
mpc0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),
nqc0 = numroc(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),

ilcm, indxg2p, and numroc are ScaLAPACK tool functions;
myrow, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

c (local).
On exit, sub(C) is overwritten by the Q * sub(C) if *side* = 'L',
or sub(C) * Q if *side* = 'R'.

p?larfb

Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.

Syntax

```
call pslarfb (side, trans, direct, storev, m, n, k, v, iv, jv, descv, t,c, ic,
    jc, descc, work)
call pdlarfb (side, trans, direct, storev, m, n, k, v, iv, jv, descv, t,c, ic,
    jc, descc, work)
call pclarfb (side, trans, direct, storev, m, n, k, v, iv, jv, descv, t,c, ic,
    jc, descc, work)
call pzlarfb (side, trans, direct, storev, m, n, k, v, iv, jv, descv, t,c, ic,
    jc, descc, work)
```

Description

This routine applies a real/complex block reflector Q or its transpose Q^T/conjugate transpose Q^H to a real/complex distributed *m*-by-*n* matrix sub(C) = C(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1) from the left or the right.

Input Parameters

side (global).CHARACTER.

if *side* = 'L': apply Q or Q^T for real flavors/Q^H for complex flavors from the Left;

if *side* = 'R': apply Q or Q^T for real flavors/Q^H for complex flavors from the Right.

trans (global).CHARACTER.

if *trans*= 'N': No transpose, apply Q;

for real flavors, if *trans*= 'T': Transpose, apply Q^T
for complex flavors, if *trans*= 'C': Conjugate transpose, apply Q^H;

direct (global) CHARACTER.
Indicates how Q is formed from a product of elementary reflectors.

if *direct* = 'F': $Q = H(1)\ H(2) \ldots H(k)$ (Forward)
if *direct* = 'B': $Q = H(k) \ldots H(2)\ H(1)$ (Backward)

storev (global) CHARACTER.
Indicates how the vectors that define the elementary reflectors are stored:
if *storev* = 'C': Columnwise
if *storev* = 'R': Rowwise.

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(C). ($m \geq 0$).

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(C). ($n \geq 0$).

k (global) INTEGER.
The order of the matrix T.

v (local).
REAL for pslarfb
DOUBLE PRECISION for pdlarfb
COMPLEX for pclarfb
COMPLEX*16 for pzlarfb.
Pointer into the local memory to an array of DIMENSION (*lld_v*, $LOCc$(*jv*+*k*-1)) if

storev = 'C', (*lld_v*, $LOCc$(*jv*+*m*-1)) if *storev* = 'R' and

side = 'L', (*lld_v*, $LOCc$(*jv*+*n*-1)) if *storev* = 'R' and

side = 'R'. It contains the local pieces of the distributed vectors V representing the Householder transformation.

If *storev* = 'C' and *side* = 'L', *lld_v* ≥max(1,$LOCr$(*iv*+*m*-*1*));

if *storev* = 'C' and *side* = 'R', *lld_v* ≥ max(1,$LOCr$(*iv*+*n*-*1*));

if *storev* = 'R', *lld_v* ≥ $LOCr$(*jv*+*k*-1).

iv, *jv* (global) INTEGER. The row and column indices in the global array V indicating the first row and the first column of the submatrix sub(V), respectively.

descv (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *V*.

c (local).
REAL for pslarfb
DOUBLE PRECISION for pdlarfb
COMPLEX for pclarfb
COMPLEX*16 for pzlarfb.
Pointer into the local memory to an array of DIMENSION (*lld_c*, *LOCc*(*jc*+*n*-1)), containing the local pieces of sub(*C*).

ic,jc (global) INTEGER. The row and column indices in the global array *C* indicating the first row and the first column of the submatrix sub(*C*), respectively.

descc (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local).

REAL for pslarfb
DOUBLE PRECISION for pdlarfb
COMPLEX for pclarfb
COMPLEX*16 for pzlarfb.
Workspace array, DIMENSION (*lwork*).

```
If storev = 'C',
  if side = 'L',
    lwork ≥ ( nqc0 + mpc0 ) * k
  else if side = 'R',
    lwork ≥ ( nqc0 + max( npv0 + numroc( numroc( n + icoffc,
              nb_v, 0, 0, npcol ), nb_v, 0, 0, lcmq ),
              mpc0 ) ) * k

  end if
else if storev = 'R',
   if side = 'L',
    lwork ≥ ( mpc0 + max( mqv0 + numroc( numroc( m+iroffc,
              mb_v, 0, 0, nprow ), mb_v, 0, 0, lcmp ),
               nqc0 ) ) * k

   else if side = 'R',
    lwork ≥ ( mpc0 + nqc0 ) * k
   end if
 end if,
```

where *lcmq* = *lcm* / *npcol* with lcm = iclm(*nprow*, *npcol*),

iroffv = mod(*iv*-1, *mb_v*), *icoffv* = mod(*jv*-1, *nb_v*),
ivrow = indxg2p(*iv*, *mb_v*, *myrow*, *rsrc_v*, *nprow*),
ivcol = indxg2p(*jv*, *nb_v*, *mycol*, *csrc_v*, *npcol*),
MqV0 = numroc(*m*+*icoffv*, *nb_v*, *mycol*, *ivcol*, *npcol*),
NpV0 = numroc(*n*+*iroffv*, *mb_v*, *myrow*, *ivrow*, *nprow*),

iroffc = mod(*ic*-1, *mb_c*), *icoffc* = mod(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),
iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),
MpC0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),
NpC0 = numroc(*n*+*icoffc*, *mb_c*, *myrow*, *icrow*, *nprow*),
NqC0 = numroc(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),

ilcm, indxg2p, and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

c (local).
On exit, sub(*C*) is overwritten by the *Q* * sub(*C*), or *Q'* *sub(*C*) or sub(*C*)**Q* or sub(*C*)**Q'*.

p?larfc

Applies the conjugate transpose of an elementary reflector to a general matrix.

Syntax

```
call pclarfc (side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pzlarfc (side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
```

Description

This routine applies a complex elementary reflector Q^H to a complex *m*-by-*n* distributed matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1), from either the left or the right. *Q* is represented in the form

Q = *i* - *tau* * *v* * *v'*,

where *tau* is a complex scalar and *v* is a complex vector.

If *tau* = 0, then Q is taken to be the unit matrix.

Input Parameters

side (global).CHARACTER.

if *side* = 'L': form Q^H*sub (C) ;

if *side* = 'R': form sub (C)*Q^H .

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(C). ($m \geq 0$).

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(C). ($n \geq 0$).

v (local).

COMPLEX for pclarfc
COMPLEX*16 for pzlarfc.
Pointer into the local memory to an array of DIMENSION (*lld_v*,*) containing the local pieces of the distributed vectors *v* representing the Householder transformation Q,

v(*iv*:*iv*+*m*-1, *jv*) if *side* = 'L' and *incv* = 1,
v(*iv*, *jv*:*jv*+*m*-1) if *side* = 'L' and *incv* = *m_v*,
v(*iv*:*iv*+*n*-1, *jv*) if *side* = 'R' and *incv* = 1,
v(*iv*, *jv*:*jv*+*n*-1) if *side* = 'R' and *incv* = *m_v*.

The vector *v* is the representation of Q. *v* is not used if *tau* = 0.

iv,jv (global) INTEGER. The row and column indices in the global array V indicating the first row and the first column of the submatrix sub(V), respectively.

descv (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix V.

incv (global) INTEGER.
The global increment for the elements of *v*. Only two values of *incv* are supported in this version, namely 1 and *m_v*.
incv must not be zero.

tau (local)
COMPLEX for pclarfc
COMPLEX*16 for pzlarfc.
Array, DIMENSION *LOCc*(*jv*) if *incv* = 1, and *LOCr*(*iv*) otherwise. This array contains the Householder scalars related to the Householder vectors.

tau is tied to the distributed matrix *V*.

c (local).
COMPLEX for pclarfc
COMPLEX*16 for pzlarfc.
Pointer into the local memory to an array of DIMENSION (*lld_c*, *LOCc*(*jc*+*n*-1)), containing the local pieces of sub(*C*).

ic,jc (global) INTEGER. The row and column indices in the global array *C* indicating the first row and the first column of the submatrix sub(*C*), respectively.

descc (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local).

COMPLEX for pclarfc
COMPLEX*16 for pzlarfc.
Workspace array, DIMENSION (*lwork*).

```
If incv = 1,
   if side = 'L',
      if ivcol = iccol,
         lwork ≥ nqc0
      else
         lwork ≥ mpc0 + max( 1, nqc0 )
      end if
    else if side = 'R',
      lwork ≥ nqc0 + max( max( 1, mpc0 ), numroc( numroc(
             n+icoffc,nb_v,0,0,npcol ),nb_v,0,0,lcmq ) )
      end if
   else if incv = m_v,
      if side = 'L',
         lwork ≥ mpc0 + max( max( 1, nqc0 ), numroc( numroc(
                m+iroffc,mb_v,0,0,nprow ),mb_v,0,0,lcmp ) )
      else if side = 'R',
         if ivrow = icrow,
            lwork ≥ mpc0
```

```
            else
               lwork ≥ nqc0 + max( 1, mpc0 )
            end if
         end if
      end if,
```

where *lcm* is the least common multiple of *nprow* and *npcol* and
lcm = ilcm(*nprow*, *npcol*), lcmp = *lcm* / *nprow*,
lcmq = *lcm* / *npcol*,

iroffc = mod(*ic*-1, *mb_c*), *icoffc* = mod(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),
iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),
mpc0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),
nqc0 = numroc(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),

ilcm, indxg2p, and numroc are ScaLAPACK tool functions;
myrow, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

c (local). On exit, sub(C) is overwritten by the Q^H * sub(C) if *side* = 'L', or sub(C) * Q^H if *side* = 'R'.

p?larfg

Generates an elementary reflector (Householder matrix).

Syntax

```
call pslarfg (n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
call pdlarfg (n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
call pclarfg (n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
call pzlarfg (n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
```

Description

This routine generates a real/complex elementary reflector H of order *n*, such that

$$H * \mathrm{sub}(X) = H * \begin{pmatrix} x(iax, jax) \\ x \end{pmatrix} = \begin{pmatrix} alpha \\ 0 \end{pmatrix}, \quad H' * H = i,$$

where *alpha* is a scalar (a real scalar - for complex flavors), and sub(*X*) is an (*n*-1)-element real/complex distributed vector *X*(*ix*:*ix*+*n*-2, *jx*) if *incx* = 1 and *X*(*ix*, *jx*:*jx*+*n*-2) if *incx* = *descx*(*m_*). *H* is represented in the form

$$H = I - tau * \begin{pmatrix} 1 \\ v \end{pmatrix} * (1\ v'),$$

where *tau* is a real/complex scalar and *v* is a real/complex (*n*-1)-element vector. Note that *H* is not Hermitian.

If the elements of sub(*X*) are all zero (and *X*(*iax*, *jax*) is real for complex flavors), then *tau* = 0 and *H* is taken to be the unit matrix.

Otherwise $1 \le \mathrm{real}(tau) \le 2$ and $\mathrm{abs}(tau-1) \le 1$.

Input Arguments

n (global) INTEGER.
The global order of the elementary reflector. $n \ge 0$.

iax, jax (global) INTEGER.
The global row and column indices in *x* of *X*(*iax*, *jax*).

x (local).

REAL for pslarfg
DOUBLE PRECISION for pdlarfg
COMPLEX for pclarfg
COMPLEX*16 for pzlarfg.

Pointer into the local memory to an array of DIMENSION (*lld_x*, *). This array contains the local pieces of the distributed vector sub(*X*). Before entry, the incremented array sub(*X*) must contain vector *x*.

ix,jx (global) INTEGER.
The row and column indices in the global array *X* indicating the first row and the first column of sub(*X*), respectively.

descx (global and local) INTEGER.
Array of DIMENSION (*dlen_*). The array descriptor for the distributed matrix *X*.

incx (global) INTEGER.

The global increment for the elements of `x`. Only two values of `incx` are supported in this version, namely 1 and `m_x`.
`incx` must not be zero.

Output Arguments

alpha (local)

`REAL` for `pslafg`
`DOUBLE PRECISION` for `pdlafg`
`COMPLEX` for `pclafg`
`COMPLEX*16` for `pzlafg`.

On exit, *alpha* is computed in the process scope having the vector sub(*X*).

x (local).

On exit, it is overwritten with the vector v.

tau (local).

`REAL` for `pslarfg`
`DOUBLE PRECISION` for `pdlarfg`
`COMPLEX` for `pclarfg`
`COMPLEX*16` for `pzlarfg`.

Array, `DIMENSION` *LOCc*(*jx*) if *incx* = 1, and *LOCr*(*ix*) otherwise. This array contains the Householder scalars related to the Householder vectors.

tau is tied to the distributed matrix *X*.

p?larft

Forms the triangular vector T of a block reflector $H=I-VTV^H$.

Syntax

```
call pslarft (direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pdlarft (direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pclarft (direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pzlarft (direct, storev, n, k, v, iv, jv, descv, tau, t, work)
```

Description

This routine forms the triangular factor T of a real/complex block reflector H of order *n*, which is defined as a product of *k* elementary reflectors.

If *direct* = 'F', $H = H(1)\ H(2)\ ...\ H(k)$ and T is upper triangular;
If *direct* = 'B', $H = H(k)\ ...\ H(2)\ H(1)$ and T is lower triangular.

If *storev* = 'C', the vector which defines the elementary reflector $H(i)$ is stored in the *i*-th column of the distributed matrix V, and

$H = I - V * T * V'$

If *storev* = 'R', the vector which defines the elementary reflector $H(i)$ is stored in the *i*-th row of the distributed matrix V, and

$H = I - V' * T * V$.

Input Arguments

direct (global) `CHARACTER*1.`
Specifies the order in which the elementary reflectors are multiplied to form the block reflector:

if *direct* = 'F': $H = H(1)\ H(2)\ ...\ H(k)$ (Forward)

if *direct* = 'B': $H = H(k)\ ...\ H(2)\ H(1)$ (Backward).

storev (global) `CHARACTER*1.`
Specifies how the vectors that define the elementary reflectors are stored (See *Application Notes* below):

if *storev* = 'C': columnwise;
if *storev* = 'R': rowwise.

n (global) `INTEGER.`
The order of the block reflector H. $n \geq 0$.

k (global) `INTEGER.`
The order of the triangular factor T (= the number of elementary reflectors).
$1 \leq k \leq mb\_v$ (= nb_v).

v `REAL` for `pslarft`
`DOUBLE PRECISION` for `pdlarft`
`COMPLEX` for `pclarft`

COMPLEX*16 for pzlarft.
Pointer into the local memory to an array of local DIMENSION (*LOCr*(iv+n-1), *LOCc*(jv+k-1))

if *storev* = 'C', and (*LOCr*(iv+k-1), *LOCc*(jv+n-1)) if *storev* = 'R'. The distributed matrix *V* contains the Householder vectors. (See *Application Notes* below).

iv, jv (global) INTEGER. The row and column indices in the global array *v* indicating the first row and the first column of the submatrix sub(*V*), respectively.

descv (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *V*.

tau (local)
REAL for pslarft
DOUBLE PRECISION for pdlarft
COMPLEX for pclarft
COMPLEX*16 for pzlarft.
Array, DIMENSION *LOCr*(iv+k-1) if *incv* = *m_v*, and *LOCc*(jv+k-1) otherwise. This array contains the Householder scalars related to the Householder vectors.

tau is tied to the distributed matrix *V*.

work (local).

REAL for pslarft
DOUBLE PRECISION for pdlarft
COMPLEX for pclarft
COMPLEX*16 for pzlarft.
Workspace array, DIMENSION (*k*(k-1)/2*).

Output Arguments

v REAL for pslarft
DOUBLE PRECISION for pdlarft
COMPLEX for pclarft
COMPLEX*16 for pzlarft.

t (local)

REAL for pslarft
DOUBLE PRECISION for pdlarft
COMPLEX for pclarft
COMPLEX*16 for pzlarft.
Array, DIMENSION (*nb_v, nb_v*) if *storev* = 'Col', and (*mb_v, mb_v*) otherwise. It

contains the *k*-by-*k* triangular factor of the block reflector associated with *v*. If *direct* = 'F', *t* is upper triangular;
if *direct* = 'B', *t* is lower triangular.

Application Notes

The shape of the matrix *V* and the storage of the vectors that define the *H*(*i*) is best illustrated by the following example with *n* = 5 and *k* = 3. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

direct = 'F' and *storev* = 'C':

$$V(iv:iv+n-1,\ jv:jv+k-1) = \begin{bmatrix} 1 & & \\ v1 & 1 & \\ v1 & v2 & 1 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \end{bmatrix}$$

direct = 'F' and *storev* = 'R':

$$V(iv:iv+k-1,\ jv:jv+n-1) = \begin{bmatrix} 1 & v1 & v1 & v1 & v1 \\ & 1 & v2 & v2 & v2 \\ & & 1 & v3 & v3 \end{bmatrix}$$

direct = 'B' and *storev* = 'C':

$$V(iv:iv+n-1,\ jv:jv+k-1) = \begin{bmatrix} v1 & v2 & v3 \\ v1 & v2 & v3 \\ 1 & v2 & v3 \\ & 1 & v3 \\ & & 1 \end{bmatrix}$$

direct = 'B' and *storev* = 'R':

$$V(iv:iv+k-1,\ jv:jv+n-1) = \begin{bmatrix} v1 & v1 & 1 & & \\ v2 & v2 & v2 & 1 & \\ v3 & v3 & v3 & v3 & 1 \end{bmatrix}$$

p?larz

Applies an elementary reflector as returned by `p?tzrzf` *to a general matrix.*

Syntax

```
call pslarz (side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc,
    work)
call pdlarz (side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc,
    work)
```

```
call pclarz (side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc,
    work)
call pzlarz (side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc,
    work)
```

Description

This routine applies a real/complex elementary reflector Q (or Q^T) to a real/complex *m*-by-*n* distributed matrix sub(C) = C(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1), from either the left or the right. Q is represented in the form

Q = I - *tau* * *v* * *v'*,

where *tau* is a real/complex scalar and *v* is a real/complex vector.

If *tau* = 0, then Q is taken to be the unit matrix.

Q is a product of *k* elementary reflectors as returned by p?tzrzf.

Input Arguments

side (global) CHARACTER.

if *side* = 'L': form Q* sub(C) ,

if *side* = 'R': form sub (C)*Q, Q=Q^T (for real flavors).

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(C). ($m \geq 0$).

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(C). ($n \geq 0$).

l (global). INTEGER.
The columns of the distributed submatrix sub(A) containing the meaningful part of the Householder reflectors. If *side* = 'L', $m \geq l \geq 0$, if *side* = 'R', $n \geq l \geq 0$.

v (local).

REAL for pslarz
DOUBLE PRECISION for pdlarz
COMPLEX for pclarz
COMPLEX*16 for pzlarz.

Pointer into the local memory to an array of DIMENSION (*lld_v*,*) containing the local pieces of the distributed vectors *v* representing the Householder transformation *Q*,

v(*iv*:*iv*+*l*-1, *jv*) if *side* = 'L' and *incv* = 1,
v(*iv*, *jv*:*jv*+*l*-1) if *side* = 'L' and *incv* = *m_v*,
v(*iv*:*iv*+*l*-1, *jv*) if *side* = 'R' and *incv* = 1,
v(*iv*, *jv*:*jv*+*l*-1) if *side* = 'R' and *incv* = *m_v*.

The vector *v* in the representation of *Q*. *v* is not used if *tau* = 0.

iv,jv (global) INTEGER. The row and column indices in the global array *V* indicating the first row and the first column of the submatrix sub(*V*), respectively.

descv (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *V*.

incv (global) INTEGER.
The global increment for the elements of *v*. Only two values of *incv* are supported in this version, namely 1 and *m_v*.

incv must not be zero.

tau (local)
REAL for pslarz
DOUBLE PRECISION for pdlarz
COMPLEX for pclarz
COMPLEX*16 for pzlarz.
Array, DIMENSION *LOCc*(*jv*) if *incv* = 1, and *LOCr*(*iv*) otherwise. This array contains the Householder scalars related to the Householder vectors.

tau is tied to the distributed matrix *V*.

c (local).
REAL for pslarz
DOUBLE PRECISION for pdlarz
COMPLEX for pclarz
COMPLEX*16 for pzlarz.
Pointer into the local memory to an array of DIMENSION (*lld_c*, *LOCc*(*jc*+*n*-1)), containing the local pieces of sub(*C*).

ic,jc (global) INTEGER. The row and column indices in the global array *C* indicating the first row and the first column of the submatrix sub(*C*), respectively.

descc (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local).

REAL for pslarz
DOUBLE PRECISION for pdlarz
COMPLEX for pclarz
COMPLEX*16 for pzlarz.
Array, DIMENSION (*lwork*)

```
If incv = 1,
  if side = 'L',
    if ivcol = iccol,
      lwork ≥ NqC0
    else
      lwork ≥ MpC0 + max( 1, NqC0 )
  end if
else if side = 'R',
  lwork ≥ NqC0 + max( max( 1, MpC0 ), numroc( numroc(
          n+icoffc,nb_v,0,0,npcol ),nb_v,0,0,lcmq ) )
 end if
else if incv = m_v,
 if side = 'L',
   lwork ≥ MpC0 + max( max( 1, NqC0 ), numroc( numroc(
m+iroffc,mb_v,0,0,nprow ),mb_v,0,0,lcmp ) )
else if side = 'R',
  if ivrow = icrow,
    lwork ≥ MpC0
  else
    lwork ≥ NqC0 + max( 1, MpC0 )
  end if
 end if
end if,
```

where *lcm* is the least common multiple of *nprow* and *npcol* and
lcm = ilcm(*nprow*, *npcol*), lcmp = *lcm* / *nprow*,
lcmq = *lcm* / *npcol*,

iroffc = mod(*ic*-1, *mb_c*), *icoffc* = mod(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),
iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),
mpc0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),
nqc0 = numroc(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),

`ilcm`, `indxg2p`, and `numroc` are ScaLAPACK tool functions; *`myrow`*, *`mycol`*, *`nprow`*, and *`npcol`* can be determined by calling the subroutine `blacs_gridinfo`.

Output Arguments

c (local). On exit, sub(*C*) is overwritten by the *Q* * sub(*C*) if *`side`* = 'L', or sub(*C*) * *Q* if *`side`* = 'R'.

p?larzb

Applies a block reflector or its transpose/conjugate-transpose as returned by `p?tzrzf` *to a general matrix.*

Syntax

```
call pslarzb (side, trans, direct, storev, m, n, k, l, v, iv, jv, descv, t, c,
    ic, jc, descc, work)
call pdlarzb (side, trans, direct, storev, m, n, k, l, v, iv, jv, descv, t, c,
    ic, jc, descc, work)
call pclarzb (side, trans, direct, storev, m, n, k, l, v, iv, jv, descv, t, c,
    ic, jc, descc, work)
call pzlarzb (side, trans, direct, storev, m, n, k, l, v, iv, jv, descv, t, c,
    ic, jc, descc, work)
```

Description

This routine applies a real/complex block reflector *Q* or its transpose Q^T (conjugate transpose Q^H for complex flavors) to a real/complex distributed *m*-by-*n* matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1) from the left or the right.

Q is a product of *k* elementary reflectors as returned by `p?tzrzf`.

Currently, only *`storev`* = 'R' and *`direct`* = 'B' are supported.

Input Arguments

side (global) `CHARACTER`.

if *`side`* = 'L': apply *Q* or Q^T (Q^H for complex flavors) from the Left;

if *side* = 'R': apply Q or Q^T (Q^H for complex flavors) from the Right.

trans (global) CHARACTER.

if *trans*= 'N': No transpose, apply Q;
if *trans*= 'T': Transpose, apply Q^T(real flavors);
if *trans*= 'C': Conjugate transpose, apply Q^H (complex flavors).

direct (global) CHARACTER.
Indicates how H is formed from a product of elementary reflectors.

if *direct* = 'F': $H = H(1)\ H(2)\ .\ .\ .\ H(k)$ (Forward, not supported yet)
if *direct* = 'B': $H = H(k)\ .\ .\ .\ H(2)\ H(1)$ (Backward)

storev (global) CHARACTER.
Indicates how the vectors that define the elementary reflectors are stored:

if *storev* = 'C': Columnwise (not supported yet).
if *storev* = 'R': Rowwise.

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(C). ($m \geq 0$).

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(C). ($n \geq 0$).

k (global) INTEGER.
The order of the matrix T. (= the number of elementary reflectors whose product defines the block reflector).

l (global) INTEGER.
The columns of the distributed submatrix sub(A) containing the meaningful part of the Householder reflectors.

If *side* = 'L', $m \geq l \geq 0$, if *side* = 'R', $n \geq l \geq 0$.

v (local).

REAL for pslarzb
DOUBLE PRECISION for pdlarzb
COMPLEX for pclarzb
COMPLEX*16 for pzlarzb.
Pointer into the local memory to an array of DIMENSION
(*lld_v*, $LOCc(jv+m-1)$) if *side* = 'L',

(*lld_v*, *LOCc*(*jv*+*m*-1)) if *side* = 'R'.
It contains the local pieces of the distributed vectors *V* representing the Householder transformation as returned by p?tzrzf.

lld_v ≥ *LOCr*(*iv*+*k*-1).

iv,jv (global) INTEGER. The row and column indices in the global array *V* indicating the first row and the first column of the submatrix sub(*V*), respectively.

descv (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *V*.

t (local)

REAL for pslarzb
DOUBLE PRECISION for pdlarzb
COMPLEX for pclarzb
COMPLEX*16 for pzlarzb.
Array, DIMENSION *mb_v* by *mb_v*.

The lower triangular matrix *T* in the representation of the block reflector.

c (local).
REAL for pslarfb
DOUBLE PRECISION for pdlarfb
COMPLEX for pclarfb
COMPLEX*16 for pzlarfb.
Pointer into the local memory to an array of DIMENSION (*lld_c*, *LOCc*(*jc*+*n*-1)).
On entry, the *m*-by-*n* distributed matrix sub(*C*).

ic,jc (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix sub(*C*), respectively.

descc (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local).

REAL for pslarzb
DOUBLE PRECISION for pdlarzb
COMPLEX for pclarzb
COMPLEX*16 for pzlarzb.
Array, DIMENSION (*lwork*).

```
If storev = 'C',
   if side = 'L',
     lwork ≥( NqC0 + MpC0 ) * k
   else if side = 'R',
     lwork≥( NqC0 + max( NpV0 + numroc( numroc( n+icoffc,
nb_v, 0, 0, npcol ), nb_v, 0, 0, lcmq ), mpc0 ) ) * k
   end if
  else if storev = 'R',
   if side = 'L',
     lwork ≥ ( mpc0 + max( mqv0 + numroc( numroc( m+iroffc,
              mb_v, 0, 0, nprow ), mb_v, 0, 0, lcmp ),
              nqc0 ) ) * k
   else if side = 'R',
     lwork ≥ ( MpC0 + NqC0 ) * k
   end if
  end if,
```

where *lcmq = lcm / npcol with lcm* = iclm(*nprow, npcol*),

iroffv = mod(*iv*-1, *mb_v*), *icoffv* = mod(*jv*-1, *nb_v*),
ivrow = indxg2p(*iv, mb_v, myrow, rsrc_v, nprow*),
ivcol = indxg2p(*jv, nb_v, mycol, csrc_v, npcol*),
MqV0 = numroc(*m+icoffv, nb_v, mycol, ivcol, npcol*),
NpV0 = numroc(*n+iroffv, mb_v, myrow, ivrow, nprow*),

iroffc = mod(*ic*-1, *mb_c*), *icoffc* = *mod*(*jc*-1, *nb_c*),
icrow = indxg2p(*ic, mb_c, myrow, rsrc_c, nprow*),
iccol = indxg2p(*jc, nb_c, mycol, csrc_c, npcol*),
MpC0 = numroc(*m+iroffc, mb_c, myrow, icrow, nprow*),
NpC0 = numroc(*n+icoffc, mb_c, myrow, icrow, nprow*),
NqC0 = numroc(*n+icoffc, nb_c, mycol, iccol, npcol*),

ilcm, indxg2p, and numroc are ScaLAPACK tool functions;
myrow, mycol, nprow, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

p?larzc

Applies (multiplies by) the conjugate transpose of an elementary reflector as returned by `p?tzrzf` *to a general matrix.*

Syntax

```
call pclarzc (side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc,
    descc, work)
call pzlarzc (side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc,
    descc, work)
```

Description

This routine applies a complex elementary reflector Q^H to a complex *m*-by-*n* distributed matrix sub(C) = C(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1), from either the left or the right. Q is represented in the form

$Q = i - tau * v * v'$,

where *tau* is a complex scalar and *v* is a complex vector.

If *tau* = 0, then Q is taken to be the unit matrix.

Q is a product of *k* elementary reflectors as returned by `p?tzrzf`.

Input Arguments

side (global) CHARACTER.

if *side* = 'L': form Q^H *sub (C);
if *side* = 'R': form sub (C)*Q^H .

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(C). ($m \geq 0$).

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(C). ($n \geq 0$).

l (global) INTEGER.

The columns of the distributed submatrix sub(*A*) containing the meaningful part of the Householder reflectors.

If *side* = 'L', $m \geq l \geq 0$, if *side* = 'R', $n \geq l \geq 0$.

v (local).

COMPLEX for pclarzc
COMPLEX*16 for pzlarzc.
Pointer into the local memory to an array of DIMENSION (*lld_v*,*) containing the local pieces of the distributed vectors *v* representing the Householder transformation *Q*,

v(*iv*:*iv*+*l*-1, *jv*) if *side* = 'L' and *incv* = 1,
v(*iv*, *jv*:*jv*+*l*-1) if *side* = 'L' and *incv* = *m_v*,
v(*iv*:*iv*+*l*-1, *jv*) if *side* = 'R' and *incv* = 1,
v(*iv*, *jv*:*jv*+*l*-1) if *side* = 'R' and *incv* = *m_v*.

The vector v in the representation of *Q*. *v* is not used if *tau* = 0.

iv,jv (global) INTEGER. The row and column indices in the global array *V* indicating the first row and the first column of the submatrix sub(*V*), respectively.

descv (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *V*.

incv (global). INTEGER.
The global increment for the elements of *v*. Only two values of *incv* are supported in this version, namely 1 and *m_v*.

incv must not be zero.

tau (local)
COMPLEX for pclarzc
COMPLEX*16 for pzlarzc.
Array, DIMENSION *LOCc*(*jv*) if *incv* = 1, and *LOCr*(*iv*) otherwise. This array contains the Householder scalars related to the Householder vectors.

tau is tied to the distributed matrix *V*.

c (local).
COMPLEX for pclarzc
COMPLEX*16 for pzlarzc.
Pointer into the local memory to an array of DIMENSION (*lld_c*, *LOCc*(*jc*+*n*-1)), containing the local pieces of sub(*C*).

ic,jc (global) INTEGER. The row and column indices in the global array *C* indicating the first row and the first column of the submatrix sub(*C*), respectively.

descc (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local).

```
If incv = 1,
  if side = 'L',
    if ivcol = iccol,
      lwork ≥ NqC0
     else
      lwork ≥ MpC0 + max( 1, NqC0 )
     end if
   else if side = 'R',
    lwork ≥ nqc0 + max( max( 1, mpc0 ), numroc( numroc(
           n+icoffc,nb_v,0,0,npcol ),nb_v,0,0,lcmq ) )
     end if
   else if incv = m_v,
      if side = 'L',
        lwork ≥ mpc0 + max( max( 1, nqc0 ), numroc( numroc(
               m+iroffc,mb_v,0,0,nprow ),mb_v,0,0,lcmp ) )
       else if side = 'R',
          if ivrow = icrow,
             lwork ≥ mpc0
           else
             lwork ≥ nqc0 + max( 1, mpc0 )
            end if
           end if
        end if,
```

where *lcm* is the least common multiple of *nprow* and *npcol* and
lcm = *ilcm*(*nprow*, *npcol*), *lcmp* = *lcm* / *nprow*,
lcmq = *lcm* / *npcol*,

iroffc = mod(*ic*-1, *mb_c*), *icoffc* = mod(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),
iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),
MpC0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),
NqC0 = numroc(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),

ilcm, *indxg2p*, and *numroc* are ScaLAPACK tool functions;

myrow, mycol, nprow, and *npcol* can be determined by calling the subroutine `blacs_gridinfo`.

p?larzt

Forms the triangular factor T of a block reflector $H=I-VTV^H$ as returned by `p?tzrzf`.

Syntax

```
call pslarzt (direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pdlarzt (direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pclarzt (direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pzlarzt (direct, storev, n, k, v, iv, jv, descv, tau, t, work)
```

Description

This routine forms the triangular factor *T* of a real/complex block reflector *H* of order > n, which is defined as a product of *k* elementary reflectors as returned by `p?tzrzf`.

If *direct* = 'F', $H = H(1)\ H(2)\ ...\ H(k)$ and *T* is upper triangular;

If *direct* = 'B', $H = H(k)\ ...\ H(2)\ H(1)$ and *T* is lower triangular.

If *storev* = 'C', the vector which defines the elementary reflector *H*(*i*) is stored in the *i*-th column of the array *v*, and

*H = i - v * t * v'*

If *storev* = 'R', the vector which defines the elementary reflector *H*(*i*) is stored in the *i*-th row of the array *v*, and

*H = i - v' * t * v*

Currently, only *storev* = 'R' and *direct* = 'B' are supported.

Input Arguments

direct (global) `CHARACTER`.
Specifies the order in which the elementary reflectors are multiplied to form the block reflector:

if *direct* = 'F': $H = H(1)\ H(2) \ldots H(k)$ (Forward, not supported yet)
if *direct* = 'B': $H = H(k) \ldots H(2)\ H(1)$ (Backward).

storev (global) CHARACTER.
Specifies how the vectors which define the elementary reflectors are stored:

if *storev* = 'C': columnwise (not supported yet);
if *storev* = 'R': rowwise.

n (global). INTEGER.
The order of the block reflector *H*. $n \geq 0$.

k (global). INTEGER.
The order of the triangular factor *T* (= the number of elementary reflectors).
$1 \leq k \leq mb\_v$ (= *nb_v*).

v REAL for pslarzt
DOUBLE PRECISION for pdlarzt
COMPLEX for pclarzt
COMPLEX*16 for pzlarzt.
Pointer into the local memory to an array of local DIMENSION
(*LOCr*(*iv*+*k*-1), *LOCc*(*jv*+*n*-1)).

The distributed matrix *V* contains the Householder vectors. See *Application Notes* below.

iv, *jv* (global) INTEGER. The row and column indices in the global array *V* indicating the first row and the first column of the submatrix sub(*V*), respectively.

descv (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *V*.

tau (local)
REAL for pslarzt
DOUBLE PRECISION for pdlarzt
COMPLEX for pclarzt
COMPLEX*16 for pzlarzt.
Array, DIMENSION *LOCr*(*iv*+*k*-1) if *incv* = *m_v*, and *LOCc*(*jv*+*k*-1) otherwise.
This array contains the Householder scalars related to the Householder vectors.

tau is tied to the distributed matrix *V*.

work (local).

REAL for pslarzt
DOUBLE PRECISION for pdlarzt
COMPLEX for pclarzt
COMPLEX*16 for pzlarzt.
Workspace array, DIMENSION (k*(k-1)/2).

Output Arguments

v REAL for pslarzt
DOUBLE PRECISION for pdlarzt
COMPLEX for pclarzt
COMPLEX*16 for pzlarzt.

t (local)

REAL for pslarzt
DOUBLE PRECISION for pdlarzt
COMPLEX for pclarzt
COMPLEX*16 for pzlarzt.
Array, DIMENSION (*mb_v*, *mb_v*). It contains the *k*-by-*k* triangular factor of the block reflector associated with *v*. *t* is lower triangular.

Application Notes

The shape of the matrix *V* and the storage of the vectors which define the *H*(*i*) is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

direct = 'F' and *storev* = 'C':

$$\begin{bmatrix} v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \end{bmatrix}$$

```
v =    .  .  .
       .  .  .
       1  .  .
          1  .
             1
```

direct = 'F' and *storev* = '*R*':

```
                 V
   ___________________________
 [ v1 v1 v1 v1 v1  . . . . 1  ]
 [ v2 v2 v2 v2 v2  . . . 1    ]
 [ v3 v3 v3 v3 v3  . . 1      ]
```

direct = 'B' and *storev* = '*C*':

```
       1
       . 1
       . . 1
       . . .
V =    . . .
     [v1 v2 v3]
     [v1 v2 v3]
     [v1 v2 v3]
     [v1 v2 v3]
     [v1 v2 v3]
```

direct = 'B' and *storev* = '*R*':

```
                  V
               ______________________
 1 . . . . [ v1 v1 v1 v1 v1 ]
 . 1 . . . [ v2 v2 v2 v2 v2 ]
 . . 1 . . [ v3 v3 v3 v3 v3 ]
```

p?lascl

Multiplies a general rectangular matrix by a real scalar defined as c_{to}/c_{from}.

Syntax

```
call pslascl (type, cfrom, cto, m, n, a, ia, ja, desca, info)
call pdlascl (type, cfrom, cto, m, n, a, ia, ja, desca, info)
call pclascl (type, cfrom, cto, m, n, a, ia, ja, desca, info)
call pzlascl (type, cfrom, cto, m, n, a, ia, ja, desca, info)
```

Description

This routine multiplies the *m*-by-*n* real/complex distributed matrix sub(*A*) denoting *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) by the real/complex scalar *cto/cfrom*. This is done without over/underflow as long as the final result *cto* * *A*(*i,j*) / *cfrom* does not over/underflow. *type* specifies that sub(*A*) may be full, upper triangular, lower triangular or upper Hessenberg.

Input Arguments

type (global) CHARACTER.

type indices of the storage type of the input distributed matrix.
if *type* = 'G': sub(*A*) is a full matrix,
if *type* = 'L': sub(*A*) is a lower triangular matrix,
if *type* = 'U': sub(*A*) is an upper triangular matrix,
if *type* = 'H': sub(*A*) is an upper Hessenberg matrix.

cfrom, cto (global)

REAL for pslascl/pclascl
DOUBLE PRECISION for pdlascl/pzlascl.

The distributed matrix sub(*A*) is multiplied by *cto/cfrom* . *A*(*i,j*) is computed without over/underflow if the final result *cto* * *A*(*i,j*) / *cfrom* can be represented without over/underflow. *cfrom* must be nonzero.

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). ($m \geq 0$).

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). ($n \geq 0$).

a (local input/local output)

REAL for pslascl
DOUBLE PRECISION for pdlascl
COMPLEX for pclascl
COMPLEX*16 for pzlascl.

Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1)).

This array contains the local pieces of the distributed matrix sub(*A*).

ia, ja (global) INTEGER.
The column and row indices in the global array *A* indicating the first row and column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER .

Array of DIMENSION (*dlen_*).The array descriptor for the distributed matrix *A*.

Output Arguments

a (local). On exit, this array contains the local pieces of the distributed matrix multiplied by *cto/cfrom*.

info (local) INTEGER.

if *info* = 0: the execution is successful.

if *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?laset

Initializes the off-diagonal elements of a matrix to α and the diagonal elements to β.

Syntax

```
call pslaset (uplo, m, n, alpha, beta, a, ia, ja, desca)
call pdlaset (uplo, m, n, alpha, beta, a, ia, ja, desca)
call pclaset (uplo, m, n, alpha, beta, a, ia, ja, desca)
call pzlaset (uplo, m, n, alpha, beta, a, ia, ja, desca)
```

Description

This routine initializes an *m*-by-*n* distributed matrix sub(A) denoting A(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) to *beta* on the diagonal and *alpha* on the offdiagonals.

Input Arguments

uplo (global) CHARACTER.

Specifies the part of the distributed matrix sub(A) to be set:

if *uplo* = 'U': upper triangular part is set; the strictly lower triangular part of sub(A) is not changed;

if *uplo* = 'L': lower triangular part is set; the strictly upper triangular part of sub(A) is not changed.

Otherwise: all of the matrix sub(A) is set.

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(A). ($m \geq 0$).

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(A). ($n \geq 0$).

alpha (global).

REAL for pslaset
DOUBLE PRECISION for pdlaset
COMPLEX for pclaset
COMPLEX*16 for pzlaset.

The constant to which the offdiagonal elements are to be set.

beta (global).

REAL for pslaset
DOUBLE PRECISION for pdlaset
COMPLEX for pclaset
COMPLEX*16 for pzlaset.

The constant to which the diagonal elements are to be set.

Output Parameters

a (local).

REAL for pslaset
DOUBLE PRECISION for pdlaset
COMPLEX for pclaset
COMPLEX*16 for pzlaset.

Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1)). This array contains the local pieces of the distributed matrix sub(*A*) to be set. On exit, the leading *m*-by-*n* submatrix sub(*A*) is set as follows:

if *uplo* = 'U', $A(ia+i-1, ja+j-1) = alpha$, $1 < i \le j-1$, $1 \le j \le n$,
if *uplo* = 'L', $A(ia+i-1, ja+j-1) = alpha$, $j+1 \le i \le m$, $1 \le j \le n$,
otherwise, $A(ia+i-1, ja+j-1) = alpha$, $1 \le i \le m$, $1 \le j \le n$, *ia*+*i*.*ne*.*ja*+*j*,
and, for all *uplo*, $A(ia+i-1, ja+i-1) = beta$, $1 \le i \le \min(m,n)$.

ia, ja (global) INTEGER.
The column and row indices in the global array *A* indicating the first row and column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER .

Array of DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

p?lasmsub

Looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero.

Syntax

```
call pslasmsub (a, desca, i, l, k, smlnum, buf, lwork)
call pdlasmsub (a, desca, i, l, k, smlnum, buf, lwork)
```

Description

This routine looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero. This routine does a global maximum and must be called by all processes.

Input Arguments

a (global)

REAL for pslasmsub
DOUBLE PRECISION for pdlasmsub
Array, DIMENSION (*desca*(*lld_*),*).
On entry, the Hessenberg matrix whose tridiagonal part is being scanned. Unchanged on exit.

desca (global and local) INTEGER.
Array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *A*.

i (global) INTEGER. The global location of the bottom of the unreduced submatrix of *A*. Unchanged on exit.

l (global) INTEGER. The global location of the top of the unreduced submatrix of *A*. Unchanged on exit.

smlnum (global)

REAL for pslasmsub
DOUBLE PRECISION for pdlasmsub
On entry, a “small number” for the given matrix. Unchanged on exit.

lwork (global) `INTEGER`.
On exit, *lwork* is the size of the work buffer.
This must be at least 2*`ceil`(`ceil`((*i*-*l*)/*hbl*)/`lcm`(*nprow*,*npcol*)). Here `lcm` is least common multiple, and *nprow* x *npcol* is the logical grid size.

Output Parameters

k (global) `INTEGER`.
On exit, this yields the bottom portion of the unreduced submatrix. This will satisfy: $l \le m \le i-1$.

buf (local).

`REAL` for `pslasmsub`
`DOUBLE PRECISION` for `pdlasmsub`
Array of size *lwork*.

p?lassq

Updates a sum of squares represented in scaled form.

Syntax

```
call pslassq (n, x, ix, jx, descx, incx, scale, sumsq)
call pdlassq (n, x, ix, jx, descx, incx, scale, sumsq)
call pclassq (n, x, ix, jx, descx, incx, scale, sumsq)
call pzlassq (n, x, ix, jx, descx, incx, scale, sumsq)
```

Description

This routine returns the values *scl* and *smsq* such that

$$scl^2 * smsq = x(1)^2 + ... + x(n)^2 + scale^2 * sumsq,$$

where $x(i) = \text{sub}(x) = x(ix + (jx-1)*descx(m\_) + (i-1)*incx)$ for `pslassq/pdlassq` and $x(i) = \text{sub}(x) = \text{abs}(x(ix + (jx-1)*descx(m\_) + (i-1)*incx)$ for `pclassq/pzlassq`.
For real routines `pslassq/pdlassq` the value of *sumsq* is assumed to be non-negative and *scl* returns the value

$scl = \max(scale, \text{abs}(x(i)))$.

For complex routines `pclassq/pzlassq` the value of *sumsq* is assumed to be at least unity and the value of *ssq* will then satisfy

$1.0 \le ssq \le sumsq + 2n$

Value *scale* is assumed to be non-negative and *scl* returns the value

$$scl = \max_i(\ scale, \text{abs}(\text{real}(x(i))),\ \ \text{abs}(\text{aimag}(x(i)))\ ).$$

For all routines `p?lassq` values *scale* and *sumsq* must be supplied in `scale` and `sumsq` respectively, and `scale` and `sumsq` are overwritten by *scl* and *ssq* respectively.

All routines `p?lassq` make only one pass through the vector sub(x).

Input Parameters

n (global) `INTEGER`. The length of the distributed vector sub(x).

x `REAL` for `pslassq`
`DOUBLE PRECISION` for `pdlassq`
`COMPLEX` for `pclassq`
`COMPLEX*16` for `pzlassq`.
The vector for which a scaled sum of squares is computed:
$x(ix + (jx-1)*m\_x + (i - 1)*incx)$, $1 \le i \le n$.

ix (global) `INTEGER`.
The row index in the global array *X* indicating the first row of sub(*X*).

jx (global) `INTEGER`.
The column index in the global array *X* indicating the first column of sub(*X*).

descx (global and local) `INTEGER` array of `DIMENSION` (*dlen_*).
The array descriptor for the distributed matrix *X*.

incx (global) `INTEGER`.
The global increment for the elements of *X*. Only two values of *incx* are supported in this version, namely 1 and *m_x*. The argument *incx* must not equal zero.

scale (local).
`REAL` for `pslassq/pclassq`
`DOUBLE PRECISION` for `pdlassq/pzlassq`.
On entry, the value *scale* in the equation above.

sumsq (local) `REAL` for `pslassq/pclassq`
`DOUBLE PRECISION` for `pdlassq/pzlassq`.
On entry, the value *sumsq* in the equation above.

Output Parameters

scale (local). On exit, *scale* is overwritten with *scl* , the scaling factor for the sum of squares.

sumsq (local).
On exit, *sumsq* is overwritten with the value *smsq,* the basic sum of squares from which *scl* has been factored out.

p?laswp

Performs a series of row interchanges on a general rectangular matrix.

Syntax

```
call pslaswp (direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
call pdlaswp (direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
call pclaswp (direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
call pzlaswp (direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
```

Description

This routine performs a series of row or column interchanges on the distributed matrix sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1). One interchange is initiated for each of rows or columns *k1* through *k2* of sub(*A*). This routine assumes that the pivoting information has already been broadcast along the process row or column. Also note that this routine will only work for *k1*-*k2* being in the same *mb* (or *nb*) block. If you want to pivot a full matrix, use p?lapiv.

Input Parameters

direc (global) CHARACTER.

Specifies in which order the permutation is applied:
= 'F' (Forward)
= 'B' (Backward).

rowcol (global) CHARACTER.
Specifies if the rows or columns are permuted:
= 'R' (Rows)
= 'C' (Columns).

n (global) INTEGER.

If *rowcol*='R', the length of the rows of the distributed matrix *A*(*, *ja*:*ja*+*n*-1) to be permuted;
If *rowcol*='C', the length of the columns of the distributed matrix *A*(*ia*:*ia*+*n*-1, *) to be permuted;

a (local) REAL for pslaswp
DOUBLE PRECISION for pdlaswp
COMPLEX for pclaswp
COMPLEX*16 for pzlaswp.
Pointer into the local memory to an array of DIMENSION (*lld_a*, *).
On entry, this array contains the local pieces of the distributed matrix to which the row/columns interchanges will be applied.

ix (global) INTEGER.
The row index in the global array *A* indicating the first row of sub(*A*).

jx (global) INTEGER.
The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *A*.

k1 (global) INTEGER. The first element of *ipiv* for which a row or column interchange will be done.

k2 (global) INTEGER. The last element of *ipiv* for which a row or column interchange will be done.

ipiv (local) INTEGER.
Array, DIMENSION *LOCr*(*m_a*)+*mb_a* for row pivoting and *LOCr*(*n_a*)+*nb_a* for column pivoting. This array is tied to the matrix *A*, *ipiv*(*k*)=*l* implies rows (or columns) *k* and *l* are to be interchanged.

Output Parameters

a (local) REAL for pslaswp
DOUBLE PRECISION for pdlaswp
COMPLEX for pclaswp
COMPLEX*16 for pzlaswp.
On exit, the permuted distributed matrix.

p?latra

Computes the trace of a general square distributed matrix.

Syntax

```
val = pslatra (n, a, ia, ja, desca)
val = pdlatra (n, a, ia, ja, desca)
val = pclatra (n, a, ia, ja, desca)
val = pzlatra (n, a, ia, ja, desca)
```

Description

This function computes the trace of an *n*-by-*n* distributed matrix sub(*A*) denoting *A*(`ia`:`ia`+`n`-1, `ja`:`ja`+`n`-1). The result is left on every process of the grid.

Input Parameters

n — (global) `INTEGER`.
The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(*A*). $n \geq 0$.

a — (local).
`REAL` for `pslatra`
`DOUBLE PRECISION` for `pdlatra`
`COMPLEX` for `pclatra`
`COMPLEX*16` for `pzlatra`.
Pointer into the local memory to an array of `DIMENSION` (*lld_a*, *LOCc*(*ja*+*n*-1)) containing the local pieces of the distributed matrix, the trace of which is to be computed.

ia, ja — (global) `INTEGER`. The row and column indices respectively in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca — (global and local) `INTEGER` array of `DIMENSION` (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

val — The value returned by the fuction.

p?latrd

Reduces the first nb *rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.*

Syntax

```
call pslatrd (uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw,
    work)
call pdlatrd (uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw,
    work)
call pclatrd (uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw,
    work)
call pzlatrd (uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw,
    work)
```

Description

This routine reduces *nb* rows and columns of a real symmetric or complex Hermitian matrix sub(A)= A(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) to symmetric/complex tridiagonal form by an orthogonal/unitary similarity transformation Q' * sub(A)* Q, and returns the matrices V and W, which are needed to apply the transformation to the unreduced part of sub(A).
If *uplo* = 'U', p?latrd reduces the last *nb* rows and columns of a matrix, of which the upper triangle is supplied;
if *uplo* = 'L', p?latrd reduces the first *nb* rows and columns of a matrix, of which the lower triangle is supplied.

This is an auxiliary routine called by p?sytrd/p?hetrd.

Input Parameters

uplo (global) CHARACTER.
Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(A) is stored:
= 'U': Upper triangular
= 'L': Lower triangular.

n (global) INTEGER.
The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(A). $n \geq 0$.

nb (global) `INTEGER`.
The number of rows and columns to be reduced.

a `REAL` for `pslatrd`
`DOUBLE PRECISION` for `pdlatrd`
`COMPLEX` for `pclatrd`
`COMPLEX*16` for `pzlatrd`.
Pointer into the local memory to an array of `DIMENSION` (*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the symmetric/Hermitian distributed matrix sub(*A*).
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

ia (global) `INTEGER`.
The row index in the global array *A* indicating the first row of sub(*A*).

ja (global) `INTEGER`.
The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) `INTEGER` array of `DIMENSION` (*dlen_*). The array descriptor for the distributed matrix *A*.

iw (global) `INTEGER`.
The row index in the global array *W* indicating the first row of sub(*W*).

jw (global) `INTEGER`.
The column index in the global array *W* indicating the first column of sub(*W*).

descw (global and local) `INTEGER` array of `DIMENSION` (*dlen_*). The array descriptor for the distributed matrix *W*.

work (local)
`REAL` for `pslatrd`
`DOUBLE PRECISION` for `pdlatrd`
`COMPLEX` for `pclatrd`
`COMPLEX*16` for `pzlatrd`.
Workspace array of `DIMENSION` (*nb_a*).

Output Parameters

a (local) On exit, if *uplo* = 'U', the last *nb* columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of sub(*A*); the elements above the diagonal with the array *tau* represent the orthogonal/unitary

matrix *Q* as a product of elementary reflectors;
if `uplo` = 'L', the first *nb* columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of sub(*A*); the elements below the diagonal with the array `tau` represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.

d (local) `REAL` for `pslatrd/pclatrd`
`DOUBLE PRECISION` for `pdlatrd/pzlatrd`.
Array, `DIMENSION` *LOCc*(`ja`+`n`-1).
The diagonal elements of the tridiagonal matrix *T*: *d*(i) = *a*(i,i). *d* is tied to the distributed matrix *A*.

e (local) `REAL` for `pslatrd/pclatrd`
`DOUBLE PRECISION` for `pdlatrd/pzlatrd`.
Array, `DIMENSION` *LOCc*(`ja`+`n`-1) if `uplo` = 'U', *LOCc*(`ja`+`n`-2) otherwise.
The off-diagonal elements of the tridiagonal matrix *T*:
e(i) = *a*(i, i +1) if `uplo` = 'U',
e(i) = *a*(i +1, i) if `uplo` = 'L'.
e is tied to the distributed matrix *A*.

tau (local) `REAL` for `pslatrd`
`DOUBLE PRECISION` for `pdlatrd`
`COMPLEX` for `pclatrd`
`COMPLEX*16` for `pzlatrd`.
Array, `DIMENSION` *LOCc*(`ja`+`n`-1).
This array contains the scalar factors `tau` of the elementary reflectors. `tau` is tied to the distributed matrix *A*.

w (local) `REAL` for `pslatrd`
`DOUBLE PRECISION` for `pdlatrd`
`COMPLEX` for `pclatrd`
`COMPLEX*16` for `pzlatrd`.
Pointer into the local memory to an array of `DIMENSION` (*lld_w*, *nb_w*).
This array contains the local pieces of the *n*-by-*nb_w* matrix *W* required to update the unreduced part of sub(*A*).

Application Notes

If `uplo` = 'U', the matrix *Q* is represented as a product of elementary reflectors

$Q = H(n)\,H(n-1)\,.\,.\,.\,H(n-nb+1)$

Each *H*(i) has the form

$H(i) = I$ - *tau*v*v'* ,

where *tau* is a real/complex scalar, and *v* is a real/complex vector with $v(i:n) = 0$ and $v(i-1) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-1, ja+i)$, and *tau* in $tau(ja+i-1)$.

If *uplo* = 'L', the matrix Q is represented as a product of elementary reflectors

$Q = H(1)\ H(2)\ .\ .\ .\ H(nb)$

Each $H(i)$ has the form

$H(i) = I$ - *tau*v*v'* ,

where *tau* is a real/complex scalar, and *v* is a real/complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and *tau* in $tau(ja+i-1)$.

The elements of the vectors *v* together form the *n*-by-*nb* matrix V which is needed, with W, to apply the transformation to the unreduced part of the matrix, using a symmetric/Hermitian rank-2*k* update of the form:

sub(A) := sub(A) - *vw'* - *wv'*.

The contents of *a* on exit are illustrated by the following examples with $n = 5$ and $nb = 2$:

if *uplo* = 'U':

$$\begin{bmatrix} a & a & a & v_4 & v_5 \\ & a & a & v_4 & v_5 \\ & & a & 1 & v_5 \\ & & & d & 1 \\ & & & & d \end{bmatrix}$$

if *uplo* = 'L':

$$\begin{bmatrix} d & & & & \\ 1 & d & & & \\ v_1 & 1 & a & & \\ v_1 & v_2 & a & a & \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

where *d* denotes a diagonal element of the reduced matrix, *a* denotes an element of the original matrix that is unchanged, and v_i denotes an element of the vector defining $H(i)$.

p?latrs

Solves a triangular system of equations with the scale factor set to prevent overflow.

Syntax

```
call pslatrs (uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx,
    scale, cnorm, work)
call pdlatrs (uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx,
    scale, cnorm, work)
call pclatrs (uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx,
    scale, cnorm, work)
call pzlatrs (uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx,
    scale, cnorm, work)
```

Description

This routine solves a triangular system of equations $Ax = \sigma b$, $A^T x = \sigma b$, or $A^H x = \sigma b$, where σ is a scale factor set to prevent overflow. The description of the routine will be extended in the future releases.

Input Parameters

uplo CHARACTER*1.
Specifies whether the matrix A is upper or lower triangular.
= 'U': Upper triangular
= 'L': Lower triangular

trans CHARACTER*1.
Specifies the operation applied to A.
= 'N': Solve $Ax = \sigma b$ (no transpose)
= 'T': Solve $A^T x = \sigma b$ (transpose)
= 'C': Solve $A^H x = \sigma b$ (conjugate transpose)

diag CHARACTER*1.
Specifies whether or not the matrix A is unit triangular.
= 'N': Non-unit triangular
= 'U': Unit triangular

normin CHARACTER*1.
Specifies whether *cnorm* has been set or not.
= 'Y': *cnorm* contains the column norms on entry;
= 'N': *cnorm* is not set on entry. On exit, the norms will be computed and stored in *cnorm*.

n INTEGER.
The order of the matrix *A*. $n \geq 0$

a REAL for pslatrs/pclatrs
DOUBLE PRECISION for pdlatrs/pzlatrs
Array, DIMENSION (*lda*, *n*). Contains the triangular matrix *A*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of the array *a* contains the upper triangular matrix, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the array *a* contains the lower triangular matrix, and the strictly upper triangular part of *a* is not referenced. If *diag* = 'U', the diagonal elements of *a* are also not referenced and are assumed to be 1.

ia,ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

x REAL for pslatrs/pclatrs
DOUBLE PRECISION for pdlatrs/pzlatrs
Array, DIMENSION (*n*). On entry, the right hand side *b* of the triangular system.

ix (global) INTEGER.The row index in the global array *x* indicating the first row of sub(*x*).

jx (global) INTEGER. The column index in the global array *x* indicating the first column of sub(*x*).

descx (global and local) INTEGER.
Array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *X*.

cnorm REAL for pslatrs/pclatrs
DOUBLE PRECISION for pdlatrs/pzlatrs.
Array, DIMENSION (*n*). If *normin* = 'Y', *cnorm* is an input argument and *cnorm* (*j*) contains the norm of the off-diagonal part of the *j*-th column of *A*. If *trans* = 'N', *cnorm* (*j*) must be greater than or equal to the infinity-norm, and if *trans* = 'T' or 'C', *cnorm*(*j*) must be greater than or equal to the 1-norm.

work (local).

REAL for pslatrs
DOUBLE PRECISION for pdlatrs
COMPLEX for pclatrs
COMPLEX*16 for pzlatrs.
Temporary workspace.

Output Parameters

x On exit, *x* is overwritten by the solution vector *x*.

scale REAL for pslatrs/pclatrs
DOUBLE PRECISION for pdlatrs/pzlatrs.
Array, DIMENSION (*lda*, *n*). The scaling factor *s* for the triangular system as described above.
If *scale* = 0, the matrix *A* is singular or badly scaled, and the vector *x* is an exact or approximate solution to $Ax = 0$.

cnorm If *normin* = 'N', *cnorm* is an output argument and *cnorm*(*j*) returns the 1-norm of the off-diagonal part of the *j*-th column of *A*.

p?latrz

Reduces an upper trapezoidal matrix to upper triangular form by means of orthogonal/unitary transformations.

Syntax

```
call pslatrz (m, n, l, a, ia, ja, desca, tau, work)
call pdlatrz (m, n, l, a, ia, ja, desca, tau, work)
call pclatrz (m, n, l, a, ia, ja, desca, tau, work)
call pzlatrz (m, n, l, a, ia, ja, desca, tau, work)
```

Description

This routine reduces the *m*-by-*n* ($m \le n$) real/complex upper trapezoidal matrix sub(*A*) = [*A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*m*-1) *A*(*ia*:*ia*+*m*-1, *ja*+*n*-*l*:*ja*+*n*-1)] to upper triangular form by means of orthogonal/unitary transformations.

The upper trapezoidal matrix sub(*A*) is factored as sub(*A*) = (*R* 0)**Z*,

where *Z* is an *n*-by-*n* orthogonal/unitary matrix and *R* is an *m*-by-*m* upper triangular matrix.

Input Parameters

m — (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). $m \geq 0$.

n — (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). $n \geq 0$.

l — (global) INTEGER.
The number of columns of the distributed submatrix sub(*A*) containing the meaningful part of the Householder reflectors. $l > 0$.

a — (local)

REAL for pslatrz
DOUBLE PRECISION for pdlatrz
COMPLEX for pclatrz
COMPLEX*16 for pzlatrz.
Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, the local pieces of the *m*-by-*n* distributed matrix sub(*A*), which is to be factored.

ia — (global) INTEGER.
The row index in the global array *A* indicating the first row of sub(*A*).

ja — (global) INTEGER.
The column index in the global array *A* indicating the first column of sub(*A*).

desca — (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *A*.

work — (local)
REAL for pslatrz
DOUBLE PRECISION for pdlatrz
COMPLEX for pclatrz
COMPLEX*16 for pzlatrz.
Workspace array, DIMENSION (*lwork*).
lwork ≥ *nq0* + max(1, *mp0*), where

iroff = mod(*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia, mb_a, myrow, rsrc_a, nprow*),
iacol = indxg2p(*ja, nb_a, mycol, csrc_a, npcol*),
mp0 = numroc(*m+iroff, mb_a, myrow, iarow, nprow*),
nq0 = numroc(*n+icoff, nb_a, mycol, iacol, npcol*),

numroc, indxg2p, and numroc are ScaLAPACK tool functions;
myrow, mycol, nprow, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

a — On exit, the leading *m*-by-*m* upper triangular part of sub(*A*) contains the upper triangular matrix *R*, and elements *n*-*l*+1 to *n* of the first *m* rows of sub(*A*), with the array *tau*, represent the orthogonal/unitary matrix *Z* as a product of *m* elementary reflectors.

tau — (local) REAL for pslatrz
DOUBLE PRECISION for pdlatrz
COMPLEX for pclatrz
COMPLEX*16 for pzlatrz.
Array, DIMENSION (*LOCr*(*ja*+*m*-1)). This array contains the scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

Application Notes

The factorization is obtained by Householder's method. The *k*-th transformation matrix, *Z*(*k*), which is used (or, in case of complex routines, whose conjugate transpose is used) to introduce zeros into the (*m* - *k* + 1)-th row of sub(*A*), is given in the form

$$Z(k) = \begin{bmatrix} I & 0 \\ 0 & T(k) \end{bmatrix},$$

where

$$T(k) = I - tau * u(k) * u(k)', \qquad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

tau is a scalar and $z(k)$ is an $(n-m)$-element vector. *tau* and $z(k)$ are chosen to annihilate the elements of the k-th row of sub(A). The scalar *tau* is returned in the k-th element of `tau` and the vector $u(k)$ in the k-th row of sub(A), such that the elements of $z(k)$ are in $a(k, m+1), \ldots, a(k, n)$. The elements of R are returned in the upper triangular part of sub(A).

Z is given by

$$Z = Z(1)\, Z(2) \ldots Z(m).$$

p?lauu2

Computes the product UU^H or L^HL, where U and L are upper or lower triangular matrices (local unblocked algorithm).

Syntax

```
call pslauu2 (uplo, n, a, ia, ja, desca)
call pdlauu2 (uplo, n, a, ia, ja, desca)
call pclauu2 (uplo, n, a, ia, ja, desca)
call pzlauu2 (uplo, n, a, ia, ja, desca)
```

Description

This routine computes the product UU' or $L'L$, where the triangular factor U or L is stored in the upper or lower triangular part of the distributed matrix sub(A)= A(`ia:ia+n-1`, `ja:ja+n-1`).

If *uplo* = 'U' or 'u', then the upper triangle of the result is stored, overwriting the factor U in sub(A).
If *uplo* = 'L' or 'l', then the lower triangle of the result is stored, overwriting the factor L in sub(A).

This is the unblocked form of the algorithm, calling BLAS Level 2 Routines. No communication is performed by this routine, the matrix to operate on should be strictly local to one process.

Input Parameters

uplo (global) CHARACTER*1.
Specifies whether the triangular factor stored in the matrix sub(*A*) is upper or lower triangular:
= 'U': Upper triangular
= 'L': Lower triangular.

n (global) INTEGER.
The number of rows and columns to be operated on, that is, the order of the triangular factor *U* or *L*. $n \geq 0$.

a (local) REAL for pslauu2
DOUBLE PRECISION for pdlauu2
COMPLEX for pclauu2
COMPLEX*16 for pzlauu2.
Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1). On entry, the local pieces of the triangular factor *U* or *L*.

ia (global) INTEGER.
The row index in the global array *A* indicating the first row of sub(*A*).

ja (global) INTEGER.
The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *A*.

Output Parameters

a (local) On exit, if *uplo* = 'U', the upper triangle of the distributed matrix sub(*A*) is overwritten with the upper triangle of the product *UU'*; if *uplo* = 'L', the lower triangle of sub(*A*) is overwritten with the lower triangle of the product *L'L*.

p?lauum

Computes the product UU^H or L^HL, where U and L are upper or lower triangular matrices.

Syntax

```
call pslauum (uplo, n, a, ia, ja, desca)
call pdlauum (uplo, n, a, ia, ja, desca)
call pclauum (uplo, n, a, ia, ja, desca)
call pzlauum (uplo, n, a, ia, ja, desca)
```

Description

This routine computes the product *UU'* or *L'L*, where the triangular factor *U* or *L* is stored in the upper or lower triangular part of the matrix sub(*A*)= *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1).

If *uplo* = 'U' or 'u', then the upper triangle of the result is stored, overwriting the factor *U* in sub(*A*).
If *uplo* = 'L' or 'l', then the lower triangle of the result is stored, overwriting the factor *L* in sub(*A*).

This is the blocked form of the algorithm, calling Level 3 PBLAS.

Input Parameters

uplo (global) CHARACTER*1.
Specifies whether the triangular factor stored in the matrix sub(*A*) is upper or lower triangular:
= 'U': Upper triangular
= 'L': Lower triangular.

n (global) INTEGER.
The number of rows and columns to be operated on, that is, the order of the triangular factor *U* or *L*. $n \geq 0$.

a (local) REAL for pslauum
DOUBLE PRECISION for pdlauum
COMPLEX for pclauum
COMPLEX*16 for pzlauum.

Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1). On entry, the local pieces of the triangular factor *U* or *L*.

ia (global) INTEGER.
The row index in the global array *A* indicating the first row of sub(*A*).

ja (global) INTEGER.
The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *A*.

Output Parameters

a (local) On exit, if *uplo* = 'U', the upper triangle of the distributed matrix sub(*A*) is overwritten with the upper triangle of the product *UU*' ; if *uplo* = 'L', the lower triangle of sub(*A*) is overwritten with the lower triangle of the product *L*'*L*.

p?lawil

Forms the Wilkinson transform.

Syntax

```
call pslawil (ii, jj, m, a, desca, h44, h33, h43h34, v)
call pdlawil (ii, jj, m, a, desca, h44, h33, h43h34, v)
```

Description

This routine gets the transform given by *h44*, *h33*, and *h43h34* into *v* starting at row *m*.

Input Parameters

ii (global) INTEGER.
Row owner of *h*(*m*+2, *m*+2).

jj (global) INTEGER.
Column owner of *h*(*m*+2, *m*+2).

m (global) INTEGER.
On entry, the location from where the transform starts (row *m*). Unchanged on exit.

a (global)
REAL for pslawil
DOUBLE PRECISION for pdlawil
Array, DIMENSION (*desca(lld_)*,*). On entry, the Hessenberg matrix. Unchanged on exit.

desca (global and local) INTEGER
Array of DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*. Unchanged on exit.

h44,

h33,

h43h34 (global)
REAL for pslawil
DOUBLE PRECISION for pdlawil
These three values are for the double shift *QR* iteration. Unchanged on exit.

Output Parameters

v (global)
REAL for pslawil
DOUBLE PRECISION for pdlawil
Array of size 3 that contains the transform on output.

p?org2l/p?ung2l

Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by p?geqlf *(unblocked algorithm).*

Syntax

```
call psorg2l (m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorg2l (m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcung2l (m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzung2l (m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine p?org2l/p?ung2l generates an *m*-by-*n* real/complex distributed matrix *Q* denoting *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) with orthonormal columns, which is defined as the last *n* columns of a product of *k* elementary reflectors of order *m*:

$Q = H(k) \ldots H(2)\, H(1)$ as returned by p?geqlf.

Input Parameters

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix *Q*. $m \geq 0$.

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix *Q*. $m \geq n \geq 0$.

k (global) INTEGER.
The number of elementary reflectors whose product defines the matrix *Q*.
$n \geq k \geq 0$.

a REAL for psorg2l
DOUBLE PRECISION for pdorg2l
COMPLEX for pcung2l
COMPLEX*16 for pzung2l.
Pointer into the local memory to an array,
DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, the *j*-th column must contain the vector that defines the elementary reflector *H*(*j*), $ja+n-k \leq j \leq ja+n-k$, as returned by p?geqlf in the *k* columns of its distributed matrix argument *A*(*ia*:*,*ja*+*n*-*k*:*ja*+*n*-1).

ia (global) INTEGER.
The row index in the global array *A* indicating the first row of sub(*A*).

ja (global) INTEGER.
The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psorg2l
DOUBLE PRECISION for pdorg2l
COMPLEX for pcung2l

COMPLEX*16 for pzung2l.
Array, DIMENSION *LOCc*(*ja*+*n*-1).
This array contains the scalar factor *tau*(*j*) of the elementary reflector *H*(*j*), as returned by p?geqlf.

work (local)
REAL for psorg2l
DOUBLE PRECISION for pdorg2l
COMPLEX for pcung2l
COMPLEX*16 for pzung2l.
Workspace array, DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least *lwork* ≥ *mpa0* + max(1, *nqa0*),
where *iroffa* = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia, mb_a, myrow, rsrc_a, nprow*),
iacol = indxg2p(*ja, nb_a, mycol, csrc_a, npcol*),
mpa0 = numroc(*m*+*iroffa, mb_a, myrow, iarow, nprow*),
nqa0 = numroc(*n*+*icoffa, nb_a, mycol, iacol, npcol*).

indxg2p and numroc are ScaLAPACK tool functions;
myrow, mycol, nprow, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, this array contains the local pieces of the *m*-by-*n* distributed matrix *Q*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
then *info* = - (*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value,
then *info* = -*i*.

p?org2r/p?ung2r

Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by `p?geqrf` *(unblocked algorithm).*

Syntax

```
call psorg2r (m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorg2r (m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcung2r (m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzung2r (m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine `p?org2r/p?ung2r` generates an *m*-by-*n* real/complex matrix *Q* denoting *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) 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 `p?geqrf`.

Input Parameters

m (global) `INTEGER`.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix *Q*. $m \geq 0$.

n (global) `INTEGER`.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix *Q*. $m \geq n \geq 0$.

k (global) `INTEGER`.
The number of elementary reflectors whose product defines the matrix *Q*. $n \geq k \geq 0$.

a `REAL` for `psorg2r`
`DOUBLE PRECISION` for `pdorg2r`
`COMPLEX` for `pcung2r`
`COMPLEX*16` for `pzung2r`.
Pointer into the local memory to an array,

DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1).
On entry, the *j*-th column must contain the vector that defines the elementary reflector *H*(*j*), *ja* ≤ *j* ≤ *ja*+*k*-1, as returned by p?geqrf in the *k* columns of its distributed matrix argument *A*(*ia*:*,*ja*:*ja*+*k*-1).

ia (global) INTEGER.
The row index in the global array *A* indicating the first row of sub(*A*).

ja (global) INTEGER.
The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psorg2r
DOUBLE PRECISION for pdorg2r
COMPLEX for pcung2r
COMPLEX*16 for pzung2r.
Array, DIMENSION *LOCc*(*ja*+*k*-1).
This array contains the scalar factor *tau*(*j*) of the elementary reflector *H*(*j*), as returned by p?geqrf. This array is tied to the distributed matrix *A*.

work (local)
REAL for psorg2r
DOUBLE PRECISION for pdorg2r
COMPLEX for pcung2r
COMPLEX*16 for pzung2r.
Workspace array, DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least *lwork* ≥ *mpa0* + max(1, *nqa0*),
where *iroffa* = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),
iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),
mpa0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nprow*),
nqa0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*).

indxg2p and numroc are ScaLAPACK tool functions;
myrow, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — On exit, this array contains the local pieces of the *m*-by-*n* distributed matrix *Q*.

work — On exit, *work*(1) returns the minimal and optimal *lwork*.

info — (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?orgl2/p?ungl2

Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by p?gelqf *(unblocked algorithm).*

Syntax

```
call psorgl2 (m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgl2 (m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcungl2 (m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungl2 (m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine p?orgl2/p?ungl2 generates a *m*-by-*n* real/complex matrix *Q* denoting *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) with orthonormal rows, which is defined as the first *m* rows of a product of *k* elementary reflectors of order *n*

$$Q = H(k) \ldots H(2)\, H(1) \text{ (for real flavors),}$$
$$Q = H(k)' \ldots H(2)'\, H(1)' \text{ (for complex flavors)}$$

as returned by p?gelqf.

Input Parameters

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix *Q*. $m \geq 0$.

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix *Q*. $n \geq m \geq 0$.

k (global) INTEGER.
The number of elementary reflectors whose product defines the matrix *Q*. $m \geq k \geq 0$.

a REAL for psorgl2
DOUBLE PRECISION for pdorgl2
COMPLEX for pcungl2
COMPLEX*16 for pzungl2.
Pointer into the local memory to an array,
DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1).
On entry, the *i*-th row must contain the vector that defines the elementary reflector *H*(*i*), $ia \leq i \leq ia+k-1$, as returned by p?gelqf in the *k* rows of its distributed matrix argument *A*(*ia*:*ia*+*k*-1,*ja*:*).

ia (global) INTEGER.
The row index in the global array *A* indicating the first row of sub(*A*).

ja (global) INTEGER.
The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psorgl2
DOUBLE PRECISION for pdorgl2
COMPLEX for pcungl2
COMPLEX*16 for pzungl2.
Array, DIMENSION *LOCr*(*ja*+*k*-1).
This array contains the scalar factors *tau*(*i*) of the elementary reflectors *H*(*i*), as returned by p?gelqf. This array is tied to the distributed matrix *A*.

work (local)
REAL for psorgl2
DOUBLE PRECISION for pdorgl2

COMPLEX for pcungl2
COMPLEX*16 for pzungl2.
Workspace array, DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least *lwork* ≥ *nqa*0 + max(1, *mpa*0),
where *iroffa* = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia, mb_a, myrow, rsrc_a, nprow*),
iacol = indxg2p(*ja, nb_a, mycol, csrc_a, npcol*),
*mpa*0 = numroc(*m+iroffa, mb_a, myrow, iarow, nprow*),
*nqa*0 = numroc(*n+icoffa, nb_a, mycol, iacol, npcol*).

indxg2p and numroc are ScaLAPACK tool functions;
myrow, mycol, nprow and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, this array contains the local pieces of the *m*-by-*n* distributed matrix *Q*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
then *info* = - (*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value,
then *info* = -*i*.

p?orgr2/p?ungr2

Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by `p?gerqf` *(unblocked algorithm).*

Syntax

```
call psorgr2 (m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgr2 (m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcungr2 (m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungr2 (m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine `p?orgr2/p?ungr2` generates an *m*-by-*n* real/complex matrix *Q* denoting *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) with orthonormal rows, which is defined as the last *m* rows of a product of *k* elementary reflectors of order *n*

$Q = H(1) H(2) . . . H(k)$ (for real flavors)
$Q = H(1)' H(2)' . . . H(k)'$ (for complex flavors)

as returned by `p?gerqf`.

Input Parameters

m — (global) `INTEGER`.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix *Q*. $m \geq 0$.

n — (global) `INTEGER`.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix *Q*. $n \geq m \geq 0$.

k — (global) `INTEGER`.
The number of elementary reflectors whose product defines the matrix *Q*. $m \geq k \geq 0$.

a — `REAL` for `psorgr2`
`DOUBLE PRECISION` for `pdorgr2`
`COMPLEX` for `pcungr2`
`COMPLEX*16` for `pzungr2`.

Pointer into the local memory to an array, DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, the *i*-th row must contain the vector that defines the elementary reflector *H*(*i*), *ia*+*m*-*k* ≤ *i* ≤ *ia*+*m*-1, as returned by p?gerqf in the *k* rows of its distributed matrix argument *A*(*ia*+*m*-*k*:*ia*+*m*-1,*ja*:*).

ia (global) INTEGER.
The row index in the global array *A* indicating the first row of sub(*A*).

ja (global) INTEGER.
The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psorgl2
DOUBLE PRECISION for pdorgl2
COMPLEX for pcungl2
COMPLEX*16 for pzungl2.
Array, DIMENSION *LOCr*(*ja*+*m*-1).
This array contains the scalar factors *tau*(*i*) of the elementary reflectors *H*(*i*), as returned by p?gerqf. This array is tied to the distributed matrix *A*.

work (local)
REAL for psorgr2
DOUBLE PRECISION for pdorgr2
COMPLEX for pcungr2
COMPLEX*16 for pzungr2.
Workspace array, DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least *lwork* ≥ *nqa*0 + max(1, *mpa*0),
where *iroffa* = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia, mb_a, myrow, rsrc_a, nprow*),
iacol = indxg2p(*ja, nb_a, mycol, csrc_a, npcol*),
*mpa*0 = numroc(*m+iroffa, mb_a, myrow, iarow, nprow*),
*nqa*0 = numroc(*n+icoffa, nb_a, mycol, iacol, npcol*).

indxg2p and numroc are ScaLAPACK tool functions;
myrow, mycol, nprow and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — On exit, this array contains the local pieces of the *m*-by-*n* distributed matrix *Q*.

work — On exit, *work*(1) returns the minimal and optimal *lwork*.

info — (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?orm2l/p?unm2l

Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by p?geqlf *(unblocked algorithm).*

Syntax

```
call psorm2l (side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
    work, lwork, info)
call pdorm2l (side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
    work, lwork, info)
call pcunm2l (side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
    work, lwork, info)
call pzunm2l (side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
    work, lwork, info)
```

Description

The routine p?orm2l/p?unm2l overwrites the general real/complex *m*-by-*n* distributed matrix sub (C)=C(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side = 'L'	*side* = 'R'
trans = 'N'	Q*sub (C)	sub (C)*Q
trans = 'T' (for real flavors)	Q^T * sub(C)	sub(C)*Q^T
trans = 'C' (for complex flavors)	Q^H * sub(C)	sub(C)*Q^H

where Q is a real orthogonal or complex unitary matrix defined as the product of *k* elementary reflectors

$$Q = H(k) \ldots H(2)\, H(1)$$

as returned by p?geqlf . Q is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side (global) CHARACTER.
= 'L': apply Q or Q^T (for real flavors)/Q^H(for complex flavors) from the left,
= 'R': apply Q or Q^T (for real flavors)/Q^H(for complex flavors) from the right.

trans (global) CHARACTER.
= 'N': apply Q (No transpose)
= 'T': apply Q^T (Transpose, for real flavors)
= 'C': apply Q^H (Conjugate transpose, for complex flavors)

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(C). $m \geq 0$.

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(C). $n \geq 0$.

k (global) INTEGER.
The number of elementary reflectors whose product defines the matrix Q.
If *side* = 'L', $m \geq k \geq 0$;
if *side* = 'R', $n \geq k \geq 0$.

a (local)
REAL for psorm2l
DOUBLE PRECISION for pdorm2l

COMPLEX for pcunm2l
COMPLEX*16 for pzunm2l.
Pointer into the local memory to an array, DIMENSION (*lld_a*, *LOCc*(*ja*+*k*-1).
On entry, the *j*-th row must contain the vector that defines the elementary reflector *H*(*j*), *ja* ≤ *j* ≤ *ja*+*k*-1, as returned by p?geqlf in the *k* columns of its distributed matrix argument *A*(*ia*:*,*ja*:*ja*+*k*-1). The argument *A*(*ia*:*,*ja*:*ja*+*k*-1) is modified by the routine but restored on exit.
If *side* = 'L', *lld_a* ≥ max(1, *LOCr*(*ia*+*m*-1)),
If *side* = 'R', *lld_a* ≥ max(1, *LOCr*(*ia*+*n*-1)).

ia (global) INTEGER.
The row index in the global array *A* indicating the first row of sub(*A*).

ja (global) INTEGER.
The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psorm2l
DOUBLE PRECISION for pdorm2l
COMPLEX for pcunm2l
COMPLEX*16 for pzunm2l.
Array, DIMENSION *LOCc*(*ja*+*n*-1). This array contains the scalar factor *tau*(*j*) of the elementary reflector *H*(*j*), as returned by p?geqlf. This array is tied to the distributed matrix *A*.

c (local)
REAL for psorm2l
DOUBLE PRECISION for pdorm2l
COMPLEX for pcunm2l
COMPLEX*16 for pzunm2l.
Pointer into the local memory to an array, DIMENSION (*lld_c*, *LOCc*(*jc*+*n*-1)).On entry, the local pieces of the distributed matrix sub (*C*).

ic (global) INTEGER.
The row index in the global array *C* indicating the first row of sub(*C*).

jc (global) INTEGER.
The column index in the global array *C* indicating the first column of sub(*C*).

descc (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *C*.

work (local)
REAL for psorm2l
DOUBLE PRECISION for pdorm2l
COMPLEX for pcunm2l
COMPLEX*16 for pzunm2l.
Workspace array, DIMENSION (*lwork*).
On exit, *work*(1) returns the minimal and optimal *lwork*.

lwork (local or global) INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least
if *side* = 'L', *lwork* ≥ *mpc0* + max(1, *nqc0*),
if *side* = 'R', *lwork* ≥ *nqc0* + max(max(1, *mpc0*), numroc (numroc(*n*+*icoffc*, *nb_a*, 0, 0, *npcol*), *nb_a*, 0, 0, *lcmq*)),

where *lcmq* = *lcm* / *npcol* with lcm = iclm(*nprow*, *npcol*),

iroffc = mod(*ic*-1, *mb_c*), *icoffc* = mod(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),
iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),
Mqc0 = numroc(*m*+*icoffc*, *nb_c*, *mycol*, *icrow*, *nprow*),
Npc0 = numroc(*n*+*iroffc*, *mb_c*, *myrow*, *iccol*, *npcol*),

ilcm, indxg2p and numroc are ScaLAPACK tool functions;
myrow, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c On exit, sub(*C*) is overwritten by *Q**sub(*C*) or *Q'**sub(*C*) or sub(*C*)**Q'* or sub(*C*)**Q*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
then *info* = - (*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value,
then *info* = -*i*.

NOTE. The distributed submatrices *A*(`ia:*, ja:*`) and *C*(`ic:ic+m-1,jc:jc+n-1`) must verify some alignment properties, namely the following expressions should be true:
If *side* = 'L', (`mb_a.eq.mb_c .AND. iroffa.eq.iroffc .AND. iarow.eq.icrow`)
If *side* = 'R', (`mb_a.eq.nb_c .AND. iroffa.eq.iroffc`).

p?orm2r/p?unm2r

Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by `p?geqrf` *(unblocked algorithm).*

Syntax

```
call psorm2r (side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
    work, lwork, info)
call pdorm2r (side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
    work, lwork, info)
call pcunm2r (side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
    work, lwork, info)
call pzunm2r (side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
    work, lwork, info)
```

Description

The routine `p?orm2r/p?unm2r` overwrites the general real/complex *m*-by-*n* distributed matrix sub (C)=C(`ic:ic+m-1,jc:jc+n-1`) with

	side = 'L'	*side* = 'R'
trans = 'N'	Q*sub (C)	sub (C)*Q
trans = 'T' (for real flavors)	Q^T * sub(C)	sub(C)*Q^T
trans = 'C' (for complex flavors)	Q^H * sub(C)	sub(C)*Q^H

where Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$$Q = H(k) \dots H(2)\, H(1)$$

as returned by p?geqrf . Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

side (global) CHARACTER.
= 'L': apply Q or Q^T (for real flavors)/Q^H(for complex flavors) from the left,
= 'R': apply Q or Q^T (for real flavors)/Q^H(for complex flavors) from the right.

trans (global) CHARACTER.
= 'N': apply Q (No transpose)
= 'T': apply Q^T (Transpose, for real flavors)
= 'C': apply Q^H (Conjugate transpose, for complex flavors)

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(C). $m \geq 0$.

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(C). $n \geq 0$.

k (global) INTEGER.
The number of elementary reflectors whose product defines the matrix Q.
If *side* = 'L', $m \geq k \geq 0$;
if *side* = 'R', $n \geq k \geq 0$.

a (local)
REAL for psorm2r
DOUBLE PRECISION for pdorm2r
COMPLEX for pcunm2r
COMPLEX*16 for pzunm2r.
Pointer into the local memory to an array, DIMENSION (*lld_a*, $LOCc(ja+k-1)$).
On entry, the *j*-th column must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by p?geqrf in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. The argument $A(ia:*, ja:ja+k-1)$ is modified by the routine but restored on exit.
If *side* = 'L', $lld\_a \geq \max(1, LOCr(ia+m-1))$,
If *side* = 'R', $lld\_a \geq \max(1, LOCr(ia+n-1))$.

ia (global) `INTEGER`.
The row index in the global array *A* indicating the first row of sub(*A*).

ja (global) `INTEGER`.
The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) `INTEGER` array of `DIMENSION` (*dlen_*).
The array descriptor for the distributed matrix *A*.

tau (local)
`REAL` for `psorm2r`
`DOUBLE PRECISION` for `pdorm2r`
`COMPLEX` for `pcunm2r`
`COMPLEX*16` for `pzunm2r`.
Array, `DIMENSION` *LOCc*(*ja*+*k*-1). This array contains the scalar factors *tau*(*j*) of the elementary reflector *H*(*j*), as returned by `p?geqrf`. This array is tied to the distributed matrix *A*.

c (local)
`REAL` for `psorm2r`
`DOUBLE PRECISION` for `pdorm2r`
`COMPLEX` for `pcunm2r`
`COMPLEX*16` for `pzunm2r`.
Pointer into the local memory to an array, `DIMENSION` (*lld_c*, *LOCc*(*jc*+*n*-1)). On entry, the local pieces of the distributed matrix sub (*C*).

ic (global) `INTEGER`.
The row index in the global array *C* indicating the first row of sub(*C*).

jc (global) `INTEGER`.
The column index in the global array *C* indicating the first column of sub(*C*).

descc (global and local) `INTEGER` array of `DIMENSION` (*dlen_*).
The array descriptor for the distributed matrix *C*.

work (local)
`REAL` for `psorm2r`
`DOUBLE PRECISION` for `pdorm2r`
`COMPLEX` for `pcunm2r`
`COMPLEX*16` for `pzunm2r`.
Workspace array, `DIMENSION` (*lwork*).

lwork (local or global) `INTEGER`.
The dimension of the array *work*.
lwork is local input and must be at least

if *side* = 'L', *lwork* ≥ *mpc0* + max(1, *nqc0*),
if *side* = 'R', *lwork* ≥ *nqc0* + max(max(1, *mpc0*), numroc (numroc(*n*+*icoffc*, *nb_a*, 0, 0, *npcol*), *nb_a*, 0, 0, *lcmq*)),

where *lcmq* = *lcm* / *npcol* with lcm = iclm(*nprow*, *npcol*),

iroffc = mod(*ic*-1, *mb_c*), *icoffc* = mod(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),
iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),
Mqc0 = numroc(*m*+*icoffc*, *nb_c*, *mycol*, *icrow*, *nprow*),
Npc0 = numroc(*n*+*iroffc*, *mb_c*, *myrow*, *iccol*, *npcol*),

ilcm, indxg2p and numroc are ScaLAPACK tool functions;
myrow, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c On exit, sub(*C*) is overwritten by *Q**sub(*C*) or *Q'**sub(*C*) or sub(*C*)**Q'* or sub(*C*)**Q*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

NOTE. The distributed submatrices *A*(*ia*:*, *ja*:*) and *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) must verify some alignment properties, namely the following expressions should be true:
If *side* = 'L', (*mb_a*.eq.*mb_c* .AND. *iroffa*.eq.*iroffc* .AND. *iarow*.eq.*icrow*)
If *side* = 'R', (*mb_a*.eq.*nb_c* .AND. *iroffa*.eq.*iroffc*).

p?orml2/p?unml2

Multiplies a general matrix by the orthogonal/unitary matrix from an LQ factorization determined by `p?gelqf` *(unblocked algorithm).*

Syntax

```
call psorml2 (side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
    work, lwork, info)
call pdorml2 (side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
    work, lwork, info)
call pcunml2 (side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
    work, lwork, info)
call pzunml2 (side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
    work, lwork, info)
```

Description

The routine `p?orml2/p?unml2` overwrites the general real/complex *m*-by-*n* distributed matrix sub $(C)=C(ic:ic+m-1,jc:jc+n-1)$ with

	side = 'L'	*side* = 'R'
trans = 'N'	Q*sub (C)	sub (C)*Q
trans = 'T' (for real flavors)	Q^T * sub(C)	sub(C)*Q^T
trans = 'C' (for complex flavors)	Q^H * sub(C)	sub(C)*Q^H

where Q is a real orthogonal or complex unitary distributed matrix defined as the product of *k* elementary reflectors

$Q = H(k) \dots H(2)\, H(1)$ (for real flavors)
$Q = H(k)' \dots H(2)'\, H(1)'$ (for complex flavors)

as returned by `p?gelqf`. Q is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side (global) CHARACTER.
= 'L': apply Q or Q^T (for real flavors)/Q^H(for complex flavors) from the left,
= 'R': apply Q or Q^T (for real flavors)/Q^H(for complex flavors) from the right.

trans (global) CHARACTER.
= 'N': apply Q (No transpose)
= 'T': apply Q^T (Transpose, for real flavors)
= 'C': apply Q^H (Conjugate transpose, for complex flavors)

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(C). $m \geq 0$.

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(C). $n \geq 0$.

k (global) INTEGER.
The number of elementary reflectors whose product defines the matrix Q.
If *side* = 'L', $m \geq k \geq 0$;
if *side* = 'R', $n \geq k \geq 0$.

a (local)
REAL for psorml2
DOUBLE PRECISION for pdorml2
COMPLEX for pcunml2
COMPLEX*16 for pzunml2.
Pointer into the local memory to an array, DIMENSION
(*lld_a*, *LOCc*(*ja*+*m*-1) if *side*='L',
(*lld_a*, *LOCc*(*ja*+*n*-1) if *side*='R',
where *lld_a* $\geq$ max (1, *LOCr*(*ia*+*k*-1)).
On entry, the *i*-th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gelqf in the *k* rows of its distributed matrix argument A(*ia*:*ia*+*k*-1, *ja*:*). The argument A(*ia*:*ia*+*k*-1, *ja*:*) is modified by the routine but restored on exit.

ia (global) INTEGER.
The row index in the global array A indicating the first row of sub(A).

ja (global) INTEGER.
The column index in the global array A indicating the first column of sub(A).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix A.

tau (local)
REAL for psorml2
DOUBLE PRECISION for pdorml2
COMPLEX for pcunml2

COMPLEX*16 for pzunml2.
Array, DIMENSION *LOCc*(*ia*+*k*-1). This array contains the scalar factors *tau*(*i*) of the elementary reflector *H*(*i*), as returned by p?gelqf. This array is tied to the distributed matrix *A*.

c (local)
REAL for psorml2
DOUBLE PRECISION for pdorml2
COMPLEX for pcunml2
COMPLEX*16 for pzunml2.
Pointer into the local memory to an array, DIMENSION (*lld_c*, *LOCc*(*jc*+*n*-1)). On entry, the local pieces of the distributed matrix sub (*C*).

ic (global) INTEGER.
The row index in the global array *C* indicating the first row of sub(*C*).

jc (global) INTEGER.
The column index in the global array *C* indicating the first column of sub(*C*).

descc (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *C*.

work (local)
REAL for psorml2
DOUBLE PRECISION for pdorml2
COMPLEX for pcunml2
COMPLEX*16 for pzunml2.
Workspace array, DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least
if *side* = 'L', *lwork* ≥ *mqc0* + max(max(1, *npc0*), numroc (numroc(*m*+*icoffc*, *mb_a*, 0, 0, *nprow*), *mb_a*, 0, 0, *lcmp*)),
if *side* = 'R', *lwork* ≥ *npc0* + max(1, *mqc0*),

where *lcmp* = *lcm* / *nprow* with lcm = iclm(*nprow*, *npcol*),

iroffc = mod(*ic*-1, *mb_c*), *icoffc* = mod(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),
iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),
Mpc0 = numroc(*m*+*icoffc*, *mb_c*, *mycol*, *icrow*, *nprow*),
Nqc0 = numroc(*n*+*iroffc*, *nb_c*, *myrow*, *iccol*, *npcol*),

`ilcm, indxg2p` and `numroc` are ScaLAPACK tool functions; *myrow, mycol, nprow,* and *npcol* can be determined by calling the subroutine `blacs_gridinfo`.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

c On exit, sub(C) is overwritten by Q*sub(C) or Q'*sub(C) or sub(C)*Q' or sub(C)*Q.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local) `INTEGER`.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
then *info* = - (*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value,
then *info* = -*i*.

NOTE. The distributed submatrices *A*(*ia*:*, *ja*:*) and *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) must verify some alignment properties, namely the following expressions should be true:
If *side* = 'L', (*nb_a*.eq.*mb_c* .AND. *icoffa*.eq.*iroffc*)
If *side* = 'R', (*nb_a*.eq.*nb_c* .AND. *icoffa*.eq.*icoffc* .AND. *iacol*.eq.*iccol*).

p?ormr2/p?unmr2

Multiplies a general matrix by the orthogonal/unitary matrix from an RQ factorization determined by `p?gerqf` *(unblocked algorithm).*

Syntax

```
call psormr2 (side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
    work, lwork, info)
call pdormr2 (side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
    work, lwork, info)
call pcunmr2 (side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
    work, lwork, info)
call pzunmr2 (side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
    work, lwork, info)
```

Description

The routine `p?ormr2/p?unmr2` overwrites the general real/complex *m*-by-*n* distributed matrix sub $(C)=C(ic:ic+m-1,jc:jc+n-1)$ with

	side = 'L'	*side* = 'R'
trans = 'N'	Q*sub (C)	sub (C)*Q
trans = 'T' (for real flavors)	Q^T * sub(C)	sub(C)*Q^T
trans = 'C' (for complex flavors)	Q^H * sub(C)	sub(C)*Q^H

where Q is a real orthogonal or complex unitary distributed matrix defined as the product of *k* elementary reflectors

$Q = H(1)\ H(2)\ ...\ H(k)$ (for real flavors)
$Q = H(1)'\ H(2)'\ ...\ H(k)'$ (for complex flavors)

as returned by `p?gerqf` . Q is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side (global) CHARACTER.
= 'L': apply Q or Q^T (for real flavors)/Q^H(for complex flavors) from the left,
= 'R': apply Q or Q^T (for real flavors)/Q^H(for complex flavors) from the right.

trans (global) CHARACTER.
= 'N': apply Q (No transpose)
= 'T': apply Q^T (Transpose, for real flavors)
= 'C': apply Q^H (Conjugate transpose, for complex flavors)

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(C). $m \geq 0$.

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(C). $n \geq 0$.

k (global) INTEGER.
The number of elementary reflectors whose product defines the matrix Q.
If *side* = 'L', $m \geq k \geq 0$;
if *side* = 'R', $n \geq k \geq 0$.

a (local)
REAL for psormr2
DOUBLE PRECISION for pdormr2
COMPLEX for pcunmr2
COMPLEX*16 for pzunmr2.
Pointer into the local memory to an array, DIMENSION
(*lld_a*, $LOCc$(*ja*+*m*-1) if *side*='L',
(*lld_a*, $LOCc$(*ja*+*n*-1) if *side*='R',
where *lld_a* ≥ max (1, $LOCr$(*ia*+*k*-1)).
On entry, the *i*-th row must contain the vector that defines the elementary reflector $H(i)$, *ia* ≤ *i* ≤ *ia*+*k*-1, as returned by p?gerqf in the *k* rows of its distributed matrix argument A(*ia*:*ia*+*k*-1, *ja*:*). The argument A(*ia*:*ia*+*k*-1, *ja*:*) is modified by the routine but restored on exit.

ia (global) INTEGER.
The row index in the global array A indicating the first row of sub(A).

ja (global) INTEGER.
The column index in the global array A indicating the first column of sub(A).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix A.

tau (local)
REAL for psormr2
DOUBLE PRECISION for pdormr2
COMPLEX for pcunmr2

COMPLEX*16 for pzunmr2.
Array, DIMENSION *LOCc*(*ia*+*k*-1). This array contains the scalar factors *tau*(*i*) of the elementary reflector *H*(*i*), as returned by p?gerqf. This array is tied to the distributed matrix *A*.

c
(local)
REAL for psormr2
DOUBLE PRECISION for pdormr2
COMPLEX for pcunmr2
COMPLEX*16 for pzunmr2.
Pointer into the local memory to an array, DIMENSION (*lld_c*, *LOCc*(*jc*+*n*-1)). On entry, the local pieces of the distributed matrix sub (*C*).

ic
(global) INTEGER.
The row index in the global array *C* indicating the first row of sub(*C*).

jc
(global) INTEGER.
The column index in the global array *C* indicating the first column of sub(*C*).

descc
(global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *C*.

work
(local)
REAL for psormr2
DOUBLE PRECISION for pdormr2
COMPLEX for pcunmr2
COMPLEX*16 for pzunmr2.
Workspace array, DIMENSION (*lwork*).

lwork
(local or global) INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least
if *side* = 'L', *lwork* ≥ *mpc0* + max(max(1, *nqc0*), numroc (numroc(*m*+*iroffc*, *mb_a*, 0, 0, *nprow*), *mb_a*, 0, 0, *lcmp*)),
if *side* = 'R', *lwork* ≥ *nqc0* + max(1, *mpc0*),

where *lcmp* = *lcm* / *nprow* with lcm = iclm(*nprow*, *npcol*),

iroffc = mod(*ic*-1, *mb_c*), *icoffc* = mod(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),
iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),
Mpc0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),
Nqc0 = numroc(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),

`ilcm, indxg2p` and `numroc` are ScaLAPACK tool functions; *myrow, mycol, nprow,* and *npcol* can be determined by calling the subroutine `blacs_gridinfo.`

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

c On exit, sub(C) is overwritten by Q*sub(C) or Q'*sub(C) or sub(C)*Q' or sub(C)*Q.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

NOTE. The distributed submatrices *A*(*ia*:*, *ja*:*) and *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) must verify some alignment properties, namely the following expressions should be true:
If *side*='L',(*nb_a*.eq.*mb_c* . AND. *icoffa*.eq.*iroffc*)
If *side* = 'R', (*nb_a*.eq.*nb_c* .AND. *icoffa*.eq.*icoffc* .AND. *iacol*.eq.*iccol*).

p?pbtrsv

Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a banded matrix computed by `p?pbtrf`.

Syntax

```
call pspbtrsv (uplo, trans, n, bw, nrhs, a, ja, desca, b, ib, descb, af,
    laf, work, lwork, info)
call pdpbtrsv (uplo, trans, n, bw, nrhs, a, ja, desca, b, ib, descb, af,
    laf, work, lwork, info)
call pcpbtrsv (uplo, trans, n, bw, nrhs, a, ja, desca, b, ib, descb, af,
    laf, work, lwork, info)
call pzpbtrsv (uplo, trans, n, bw, nrhs, a, ja, desca, b, ib, descb, af,
    laf, work, lwork, info)
```

Description

The routine `p?pbtrsv` solves a banded triangular system of linear equations

$$A(1:n, ja:ja+n-1)*X = B(jb:jb+n-1, 1:nrhs)$$

or

$$A(1:n, ja:ja+n-1)^T *X = B(jb:jb+n-1, 1:nrhs) \text{ for real flavors,}$$
$$A(1:n, ja:ja+n-1)^H *X = B(jb:jb+n-1, 1:nrhs) \text{ for complex flavors,}$$

where $A(1:n, ja:ja+n-1)$ is a banded triangular matrix factor produced by the Cholesky factorization code `p?pbtrf` and is stored in $A(1:n, ja:ja+n-1)$ and *af*. The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to *uplo*, and the choice of solving $A(1:n, ja:ja+n-1)$ or $A(1:n, ja:ja+n-1)^T$ for real flavors and $A(1:n, ja:ja+n-1)^H$ for complex flavors respectively is dictated by the user by the parameter *trans*.

Routine `p?pbtrf` must be called first.

Input Parameters

uplo (global) `CHARACTER`. Must be `'U'` or `'L'`.

If *uplo* = `'U'`, upper triangle of $A(1:n, ja:ja+n-1)$ is stored;
If *uplo* = `'L'`, lower triangle of $A(1:n, ja:ja+n-1)$ is stored.

trans (global) `CHARACTER`. Must be `'N'` or `'T'` or `'C'`.

If *trans* = 'N', solve with $A(1:n, ja:ja+n-1)$;

If *trans* = 'T' or 'C' for real flavors, solve with $A(1:n, ja:ja+n-1)^T$.

If *trans* = 'C' for complex flavors, solve with conjugate_transpose ($A(1:n, ja:ja+n-1)$).

n (global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $A(1:n, ja:ja+n-1)$. $n \geq 0$.

bw (global) INTEGER. The number of subdiagonals in 'L' or 'U', $0 \leq bw \leq n-1$.

nrhs (global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix $B(jb:jb+n-1, 1:nrhs)$; $nrhs \geq 0$.

a (local)

REAL for pspbtrsv
DOUBLE PRECISION for pdpbtrsv
COMPLEX for pcpbtrsv
COMPLEX*16 for pzpbtrsv.
Pointer into the local memory to an array with the first DIMENSION $lld\_a \geq (bw+1)$, stored in *desca*.

On entry, this array contains the local pieces of the *n*-by-*n* symmetric banded distributed Cholesky factor L or $L^T A(1:n, ja:ja+n-1)$.

This local portion is stored in the packed banded format used in LAPACK. Please see the *Application Notes* below and the ScaLAPACK manual for more detail on the format of distributed matrices.

ja (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.
If 1D type ($dtype\_a = 501$), then $dlen \geq 7$;
If 2D type ($dtype\_a = 1$), then $dlen \geq 9$.
Contains information on mapping of *A* to memory. Please, see ScaLAPACK manual for full description and options.

b (local)

REAL for pspbtrsv
DOUBLE PRECISION for pdpbtrsv
COMPLEX for pcpbtrsv

COMPLEX*16 for pzpbtrsv.
Pointer into the local memory to an array of local lead DIMENSION *lld_b* ≥ *nb*.
On entry, this array contains the local pieces of the right hand sides *B*(*jb*:*jb*+*n*-1, 1:*nrhs*).

ib (global) INTEGER. The row index in the global array *B* that points to the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *B*.

If 1D type (*dtype_b* = 502), then *dlen* ≥ 7;
If 2D type (*dtype_b* = 1), then *dlen* ≥ 9.
Contains information on mapping of *B* to memory. Please, see ScaLAPACK manual for full description and options.

laf (local) INTEGER. The size of user-input auxiliary Fillin space *af*.
Must be *laf* ≥ (*nb*+2**bw*)**bw* .
If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work (local)
REAL for pspbtrsv
DOUBLE PRECISION for pdpbtrsv
COMPLEX for pcpbtrsv
COMPLEX*16 for pzpbtrsv.
The array *work* is a temporary workspace array of DIMENSION *lwork*. This space may be overwritten in between calls to routines.

lwork (local or global) INTEGER. The size of the user-input workspace *work*, must be at least *lwork* ≥ *bw***nrhs*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

Output Parameters

af (local)
REAL for pspbtrsv
DOUBLE PRECISION for pdpbtrsv
COMPLEX for pcpbtrsv
COMPLEX*16 for pzpbtrsv.
The array *af* is of DIMENSION *laf*. It contains auxiliary Fillin space. Fillin is

created during the factorization routine p?pbtrf and this is stored in *af*. If a linear system is to be solved using p?pbtrs after the factorization routine, *af* must not be altered after the factorization.

b — On exit, this array contains the local piece of the solutions distributed matrix *X*.

work(1) — On exit, *work*(1) contains the minimum value of *lwork*.

info — (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

If the factorization routine and the solve routine are to be called separately to solve various sets of right-hand sides using the same coefficient matrix, the auxiliary space *af* must not be altered between calls to the factorization routine and the solve routine.

The best algorithm for solving banded and tridiagonal linear systems depends on a variety of parameters, especially the bandwidth. Currently, only algorithms designed for the case N/P >> *bw* are implemented. These algorithms go by many names, including Divide and Conquer, Partitioning, domain decomposition-type, etc.

Algorithm description: Divide and Conquer. *

The Divide and Conquer algorithm assumes the matrix is narrowly banded compared with the number of equations. In this situation, it is best to distribute the input matrix *A* one-dimensionally, with columns atomic and rows divided amongst the processes. The basic algorithm divides the banded matrix up into *P* pieces with one stored on each processor, and then proceeds in 2 phases for the factorization or 3 for the solution of a linear system.

5. **Local Phase**: The individual pieces are factored independently and in parallel. These factors are applied to the matrix creating fill-in, which is stored in a non-inspectable way in auxiliary space *af*. Mathematically, this is equivalent to reordering the matrix *A* as $P A P^T$ and then factoring the principal leading submatrix of size equal to the sum of the sizes of the matrices factored on each processor. The factors of these submatrices overwrite the corresponding parts of *A* in memory.

6. **Reduced System Phase:** A small (bw* (P-1)) system is formed representing interaction of the larger blocks and is stored (as are its factors) in the space *af*. A parallel Block Cyclic Reduction algorithm is used. For a linear system, a parallel front solve followed by an analogous backsolve, both using the structure of the factored matrix, are performed.
7. **Backsubsitution Phase:** For a linear system, a local backsubstitution is performed on each processor in parallel.

p?pttrsv

Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a tridiagonal matrix computed by `p?pttrf` .

Syntax

```
call pspttrsv (uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf,
    work, lwork, info)
call pdpttrsv (uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf,
    work, lwork, info)
call pcpttrsv (uplo, trans, n, nrhs, d, e, ja, desca, b, ib, descb, af,
    laf, work, lwork, info)
call pzpttrsv (uplo, trans, n, nrhs, d, e, ja, desca, b, ib, descb, af,
    laf, work, lwork, info)
```

Description

This routine solves a tridiagonal triangular system of linear equations

$$A(1:n, ja:ja+n-1)*X = B(jb:jb+n-1, 1:nrhs)$$

or

$$A(1:n, ja:ja+n-1)^T *X = B(jb:jb+n-1, 1:nrhs)$$ for real flavors,

$$A(1:n, ja:ja+n-1)^H *X = B(jb:jb+n-1, 1:nrhs)$$ for complex flavors,

where $A(1:n, ja:ja+n-1)$ is a tridiagonal triangular matrix factor produced by the Cholesky factorization code p?pttrf and is stored in $A(1:n, ja:ja+n-1)$ and *af*. The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to *uplo*, and the choice of solving $A(1:n, ja:ja+n-1)$ or $A(1:n, ja:ja+n-1)^T$ for real flavors and $A(1:n, ja:ja+n-1)^H$ for complex flavors respectively is dictated by the user by the parameter *trans*.

Routine `p?pttrf` must be called first.

Input Parameters

uplo (global) `CHARACTER`. Must be `'U'` or `'L'`.

If *uplo* = `'U'`, upper triangle of $A(1:n, ja:ja+n-1)$ is stored;
If *uplo* = `'L'`, lower triangle of $A(1:n, ja:ja+n-1)$ is stored.

trans (global) `CHARACTER`. Must be `'N'` or `'C'`.

If *trans* = `'N'`, solve with $A(1:n, ja:ja+n-1)$;

If *trans* = `'C'` (for complex flavors), solve with `conjugate_transpose` ($A(1:n, ja:ja+n-1)$).

n (global) `INTEGER`. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $A(1:n, ja:ja+n-1)$. $n \ge 0$.

nrhs (global) `INTEGER`. The number of right hand sides; the number of columns of the distributed submatrix $B(jb:jb+n-1, 1:nrhs)$; $nrhs \ge 0$.

d (local)

`REAL` for `pspttrsv`
`DOUBLE PRECISION` for `pdpttrsv`
`COMPLEX` for `pcpttrsv`
`COMPLEX*16` for `pzpttrsv`.

Pointer to the local part of the global vector storing the main diagonal of the matrix; must be of size ≥ *desca*(*nb_*).

e (local)

`REAL` for `pspttrsv`
`DOUBLE PRECISION` for `pdpttrsv`
`COMPLEX` for `pcpttrsv`
`COMPLEX*16` for `pzpttrsv`.

Pointer to the local part of the global vector storing the upper diagonal of the matrix; must be of size ≥ *desca*(*nb_*). Globally, *du*(*n*) is not referenced, and *du* must be aligned with *d*.

ja (global) `INTEGER`. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.
If 1D type (*dtype_a* = 501 or 502), then *dlen* ≥ 7;
If 2D type (*dtype_a* = 1), then *dlen* ≥ 9.
Contains information on mapping of *A* to memory. Please, see ScaLAPACK manual for full description and options.

b (local)

REAL for pspttrsv
DOUBLE PRECISION for pdpttrsv
COMPLEX for pcpttrsv
COMPLEX*16 for pzpttrsv.
Pointer into the local memory to an array of local lead DIMENSION *lld_b* ≥ *nb*.
On entry, this array contains the local pieces of the right hand sides *B*(*jb*:*jb*+*n*-1, 1:*nrhs*).

ib (global) INTEGER. The row index in the global array *B* that points to the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *B*.

If 1D type (*dtype_b* = 502), then *dlen* ≥ 7;
If 2D type (*dtype_b* = 1), then *dlen* ≥ 9.
Contains information on mapping of *B* to memory. Please, see ScaLAPACK manual for full description and options.

laf (local) INTEGER. The size of user-input auxiliary Fillin space *af*.
Must be *laf* ≥ (*nb*+2**bw*)**bw* .
If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work (local)
REAL for pspttrsv
DOUBLE PRECISION for pdpttrsv
COMPLEX for pcpttrsv
COMPLEX*16 for pzpttrsv.
The array *work* is a temporary workspace array of DIMENSION *lwork*. This space may be overwritten in between calls to routines.

lwork (local or global) INTEGER. The size of the user-input workspace *work*, must be at least *lwork* ≥ (10+2*min(100, *nrhs*))*npcol*+4*nrhs*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

Output Parameters

d, *e* (local).
REAL for pspttrsv
DOUBLE PRECISION for pdpttrsv
COMPLEX for pcpttrsv
COMPLEX*16 for pzpttrsv.
On exit, these arrays contain information containing the factors of the matrix.

af (local)
REAL for pspttrsv
DOUBLE PRECISION for pdpttrsv
COMPLEX for pcpttrsv
COMPLEX*16 for pzpttrsv.
The array *af* is of DIMENSION *laf*. It contains auxiliary Fillin space. Fillin is created during the factorization routine p?pbtrf and this is stored in *af*. If a linear system is to be solved using p?pttrs after the factorization routine, *af* must not be altered after the factorization.

b On exit, this array contains the local piece of the solutions distributed matrix *X*.

work(1) On exit, *work*(1) contains the minimum value of *lwork*.

info (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
then *info* = - (*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value,
then *info* = -*i*.

p?potf2

Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (local unblocked algorithm).

Syntax

```
call pspotf2 (uplo, n, a, ia, ja, desca, info)
call pdpotf2 (uplo, n, a, ia, ja, desca, info)
call pcpotf2 (uplo, n, a, ia, ja, desca, info)
call pzpotf2 (uplo, n, a, ia, ja, desca, info)
```

Description

This routine computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite distributed matrix sub $(A)=A(ia:ia+n-1, ja:ja+n-1)$.

The factorization has the form
sub $(A) = U' U$, if *uplo* = 'U', or
sub $(A) = L L'$, if *uplo* = 'L',
where U is an upper triangular matrix and L is lower triangular.

Input Parameters

uplo (global) CHARACTER.
Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix A is stored.
= 'U': Upper triangle of sub (A) is stored;
= 'L': Lower triangle of sub (A) is stored.

n (global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub (A). $n \geq 0$.

a (local)
REAL for pspotf2
DOUBLE PRECISION or pdpotf2
COMPLEX for pcpotf2
COMPLEX*16 for pzpotf2.
Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1)) containing the local pieces of the *n*-by-*n* symmetric distributed matrix sub(A) to be

factored.
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular matrix and the strictly lower triangular part of this matrix is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular matrix and the strictly upper triangular part of sub(*A*) is not referenced.

ia,ja (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the sub(*A*), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

a (local) On exit,
if *uplo* = 'U', the upper triangular part of the distributed matrix contains the Cholesky factor U;
if *uplo* = 'L', the lower triangular part of the distributed matrix contains the Cholesky factor L.

info (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
then *info* = - (*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value,
then *info* = -*i*.
> 0: if *info* = *k*, the leading minor of order *k* is not positive definite, and the factorization could not be completed.

p?rscl

Multiplies a vector by the reciprocal of a real scalar.

Syntax

```
call psrscl (n, sa, sx, ix, jx, descx, incx)
call pdrscl (n, sa, sx, ix, jx, descx, incx)
call pcsrscl (n, sa, sx, ix, jx, descx, incx)
call pzdrscl (n, sa, sx, ix, jx, descx, incx)
```

Description

This routine multiplies an *n*-element real/complex vector sub(*x*) by the real scalar 1/*a*. This is done without overflow or underflow as long as the final result sub(*x*)/*a* does not overflow or underflow.

sub(*x*) denotes *x*(*ix*:*ix*+*n*-1, *jx*:*jx*), if *incx* = 1,
and *x*(*ix*:*ix*, *jx*:*jx*+*n*-1), if *incx* = *m_x*.

Input Parameters

n (global) INTEGER.
The number of components of the distributed vector sub(*x*). $n \geq 0$.

sa REAL for psrscl/pcsrscl
DOUBLE PRECISION for pdrscl/pzdrscl.
The scalar *a* that is used to divide each component of the vector *x*. This argument must be ≥ 0, or the subroutine will divide by zero.

sx REAL for psrscl
DOUBLE PRECISION for pdrscl
COMPLEX for pcsrscl
COMPLEX*16 for pzdrscl.
Array containing the local pieces of a distributed matrix of DIMENSION of at least ((*jx*-1)**m_x* + *ix* + (*n*-1)*abs(*incx*)).
This array contains the entries of the distributed vector sub(*x*).

ix (global) INTEGER. The row index of the submatrix of the distributed matrix *X* to operate on.

jx (global) INTEGER. The column index of the submatrix of the distributed matrix *X* to operate on.

descx (global and local). INTEGER.
Array of DIMENSION 8. The array descriptor for the distributed matrix *X*.

incx (global) INTEGER.
The increment for the elements of *X*. This version supports only two values of *incx*, namely 1 and *m_x*.

Output Parameters

sx On exit, the result *x*/*a*.

p?sygs2/p?hegs2

Reduces a symmetric/Hermitian definite generalized eigenproblem to standard form, using the factorization results obtained from `p?potrf` *(local unblocked algorithm).*

Syntax

```
call pssygs2 (ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
call pdsygs2 (ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
call pchegs2 (ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
call pzhegs2 (ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
```

Description

The routine `p?sygs2/p?hegs2` reduces a real symmetric-definite or a complex Hermitian-definite generalized eigenproblem to standard form.

sub(A) denotes A(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) and sub(B) denotes B(*ib*:*ib*+*n*-1, *jb*:*jb*+*n*-1).

If *ibtype* = 1, the problem is

$$\text{sub}(A)x = \lambda\text{sub}(B)x,$$

and sub(A) is overwritten by

inv(U^T)*sub(A)*inv(U) or inv(L)*sub(A)*inv(L^T) for real flavors and
inv(U^H)*sub(A)*inv(U) or inv(L)*sub(A)*inv(L^H) for complex flavors.

If *ibtype* = 2 or 3, the problem is

$$\text{sub}(A)\text{sub}(B)x = \lambda x \quad \text{or} \quad \text{sub}(B)\text{sub}(A)x = \lambda x,$$

and sub(A) is overwritten

by U*sub(A)*U^T or L**T*sub(A)*L for real flavors and
by U*sub(A)*U^H or L**H*sub(A)*L for complex flavors.

sub(B) must have been previously factorized as U^TU or $L L^T$ (for real flavors) or as U^HU or LL^H (for complex flavors) by `p?potrf`.

Input Parameters

ibtype (global) INTEGER.
= 1: compute inv(U^T)*sub(A)*inv(U) or inv(L)*sub(A)*inv(L^T) for real subroutines and inv(U^H)*sub(A)*inv(U) or inv(L)*sub(A)*inv(L^H) for complex subroutines;
= 2 or 3: compute U*sub(A)*U^T or L^T*sub(A)*L for real subroutines and by U*sub(A)*U^H or L^H*sub(A)*L for complex subroutines.

uplo (global) CHARACTER
Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(A) is stored, and how sub(B) is factorized.
= 'U': Upper triangular of sub(A) is stored and sub(B) is factorized as U^TU (for real subroutines) or as U^HU (for complex subroutines).
= 'L': Lower triangular of sub(A) is stored and sub(B) is factorized as $L L^T$ (for real subroutines) or as $L L^H$ (for complex subroutines)

n (global) INTEGER.
The order of the matrices sub(A) and sub(B). $n \geq 0$.

a (local)
REAL for pssygs2
DOUBLE PRECISION for pdsygs2
COMPLEX for pchegs2
COMPLEX*16 for pzhegs2.
Pointer into the local memory to an array, DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the *n*-by-*n* symmetric/Hermitian distributed matrix sub(A).
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(A) contains the upper triangular part of the matrix, and the strictly lower triangular part of sub(A) is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(A) contains the lower triangular part of the matrix, and the strictly upper triangular part of sub(A) is not referenced.

ia, *ja* (global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the sub(A), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

b (local)
REAL for pssygs2
DOUBLE PRECISION for pdsygs2
COMPLEX for pchegs2

COMPLEX*16 for pzhegs2.
Pointer into the local memory to
an array, DIMENSION (*lld_b*, *LOCc*(*jb*+*n*-1)).
On entry, this array contains the local pieces of the triangular factor from the Cholesky factorization of sub(*B*) as returned by p?potrf.

ib, *jb* (global) INTEGER. The row and column indices in the global array *B* indicating the first row and the first column of the sub(*B*), respectively.

descb (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *B*.

Output Parameters

a (local) On exit, if *info* = 0, the transformed matrix is stored in the same format as sub(*A*).

info INTEGER.
= 0: successful exit.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100),
if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?sytd2/p?hetd2

Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (local unblocked algorithm).

Syntax

```
call pssytd2 (uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pdsytd2 (uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pchetd2 (uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pzhetd2 (uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

Description

The routine `p?sytd2/p?hetd2` reduces a real symmetric/complex Hermitian matrix sub(A) to symmetric/Hermitian tridiagonal form T by an orthogonal/unitary similarity transformation: Q' sub(A)$Q = T$, where sub(A) = A(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1).

Input Parameters

uplo (global) `CHARACTER`.
Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(A) is stored:
= '`U`': Upper triangular
= '`L`': Lower triangular

n (global) `INTEGER`.
The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(A). $n \geq 0$.

a (local)
`REAL` for `pssytd2`
`DOUBLE PRECISION` for `pdsytd2`
`COMPLEX` for `pchetd2`
`COMPLEX*16` for `pzhetd2`.
Pointer into the local memory to an array, `DIMENSION` (*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the *n*-by-*n* symmetric/Hermitian distributed matrix sub(A).
If *uplo* = '`U`', the leading *n*-by-*n* upper triangular part of sub(A) contains the upper triangular part of the matrix, and the strictly lower triangular part of sub(A) is not referenced. If *uplo* = '`L`', the leading *n*-by-*n* lower triangular part of sub(A) contains the lower triangular part of the matrix, and the strictly upper triangular part of sub(A) is not referenced.

ia, ja (global) `INTEGER`. The row and column indices in the global array A indicating the first row and the first column of the sub(A), respectively.

desca (global and local) `INTEGER` array, `DIMENSION` (*dlen_*). The array descriptor for the distributed matrix A.

work (local)
`REAL` for `pssytd2`
`DOUBLE PRECISION` for `pdsytd2`

COMPLEX for pchetd2
COMPLEX*16 for pzhetd2.
The array *work* is a temporary workspace array of DIMENSION *lwork*.

Output Parameters

a On exit, if *uplo* = 'U', the diagonal and first superdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first superdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors;
if *uplo* = 'L', the diagonal and first subdiagonal of *a* are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements below the first subdiagonal, with the array *tau,* represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors. See the *Application Notes* below.

d (local)
REAL for pssytd2/pchetd2
DOUBLE PRECISION for pdsytd2/pzhetd2.
Array, DIMENSION (*LOCc*(*ja*+*n*-1)).
The diagonal elements of the tridiagonal matrix *T*:
d(i) = *a*(i,i); *d* is tied to the distributed matrix *A*.

e (local)
REAL for pssytd2/pchetd2
DOUBLE PRECISION for pdsytd2/pzhetd2.
Array, DIMENSION (*LOCc*(*ja*+*n*-1)), if *uplo* = 'U', *LOCc*(*ja*+*n*-2) otherwise.
The off-diagonal elements of the tridiagonal matrix *T*:
e(i) = *a*(i,i+1) if *uplo* = 'U',
e(i) = *a*(i+1,i) if *uplo* = 'L'.
e is tied to the distributed matrix *A*.

tau (local)
REAL for pssytd2
DOUBLE PRECISION for pdsytd2
COMPLEX for pchetd2
COMPLEX*16 for pzhetd2.
Array, DIMENSION (*LOCc*(*ja*+*n*-1)).
The scalar factors of the elementary reflectors.
tau is tied to the distributed matrix *A*.

work(1) On exit, *work*(1) returns the minimal and optimal value of *lwork*.

lwork (local or global) INTEGER.
The dimension of the workspace array *work*.
lwork is local input and must be at least $lwork \geq 3n$.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

info (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100),
if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

If *uplo* = 'U', the matrix Q is represented as a product of elementary reflectors

$$Q = H(n-1) \ldots H(2)\, H(1)$$

Each H(i) has the form

$$H(i) = I - tau*v*v',$$

where *tau* is a real/complex scalar, and *v* is a real/complex vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-2,\ ja+i)$, and *tau* in $TAU(ja+i-1)$.

If *uplo* = 'L', the matrix Q is represented as a product of elementary reflectors

$$Q = H(1)\, H(2) \ldots H(n-1).$$

Each H(i) has the form
$$H(i) = I - tau*v*v',$$
where *tau* is a real/complex scalar, and *v* is a real/complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1,\ ja+i-1)$, and *tau* in $TAU(ja+i-1)$.

The contents of sub (*A*) on exit are illustrated by the following examples with $n = 5$:

if `uplo` = 'U': if `uplo` = 'L':

$$\begin{bmatrix} d & e & v_2 & v_3 & v_4 \\ & d & e & v_3 & v_4 \\ & & d & e & v_4 \\ & & & d & e \\ & & & & d \end{bmatrix} \qquad \begin{bmatrix} d & & & & \\ e & d & & & \\ v_1 & e & d & & \\ v_1 & v_2 & e & d & \\ v_1 & v_2 & v_3 & e & d \end{bmatrix}$$

where d and e denotes diagonal and off-diagonal elements of *T*, and v_i denotes an element of the vector defining *H*(i).

NOTE. The distributed submatrix sub(*A*) must verify some alignment properties, namely the following expression should be true: (*mb_a*.eq.*nb_a* . AND. *iroffa*.eq.*icoffa*) with *iroffa* = mod(*ia* - 1, *mb_a*) and *icoffa* = mod(*ja* - 1, *nb_a*).

p?trti2

Computes the inverse of a triangular matrix (local unblocked algorithm).

Syntax

```
call pstrti2 (uplo, diag, n, a, ia, ja, desca, info)
call pdtrti2 (uplo, diag, n, a, ia, ja, desca, info)
call pctrti2 (uplo, diag, n, a, ia, ja, desca, info)
call pztrti2 (uplo, diag, n, a, ia, ja, desca, info)
```

Description

This routine computes the inverse of a real/complex upper or lower triangular block matrix sub (*A*) = *A*(`ia`:`ia`+*n*-1, `ja`:`ja`+*n*-1).

This matrix should be contained in one and only one process memory space (local operation).

Input Parameters

uplo (global) `CHARACTER*1`.
Specifies whether the matrix sub (*A*) is upper or lower triangular.
= 'U': sub (*A*) is upper triangular
= 'L': sub (*A*) is lower triangular.

diag (global) `CHARACTER*1`.
Specifies whether or not the matrix *A* is unit triangular.
= 'N': sub (*A*) is non-unit triangular
= 'U': sub (*A*) is unit triangular.

n (global) `INTEGER`.
The number of rows and columns to be operated on, i.e., the order of the distributed submatrix sub(*A*). $n \geq 0$.

a (local)
`REAL` for `pstrti2`
`DOUBLE PRECISION` for `pdtrti2`
`COMPLEX` for `pctrti2`
`COMPLEX*16` for `pztrti2`.
Pointer into the local memory to an array, `DIMENSION` (*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the triangular matrix sub(*A*).
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of the matrix sub(*A*) contains the upper triangular part of the matrix, and the strictly lower triangular part of sub(*A*) is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the matrix sub(*A*) contains the lower triangular part of the matrix, and the strictly upper triangular part of sub(*A*) is not referenced.
If *diag* = 'U', the diagonal elements of sub(*A*) are not referenced either and are assumed to be 1.

ia, ja (global) `INTEGER`. The row and column indices in the global array *A* indicating the first row and the first column of the sub(*A*), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

a On exit, the (triangular) inverse of the original matrix, in the same storage format.

info INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
then *info* = - (*i**100),
if the *i*-th argument is a scalar and had an illegal value,
then *info* = -*i*.

?lamsh

Sends multiple shifts through a small (single node) matrix to maximize the number of bulges that can be sent through.

Syntax

```
call slamsh (s, lds, nbulge, jblk, h, ldh, n, ulp)
call dlamsh (s, lds, nbulge, jblk, h, ldh, n, ulp)
```

Description

This routine sends multiple shifts through a small (single node) matrix to see how small consecutive subdiagonal elements are modified by subsequent shifts in an effort to maximize the number of bulges that can be sent through. The subroutine should only be called when there are multiple shifts/bulges (*nbulge* > 1) and the first shift is starting in the middle of an unreduced Hessenberg matrix because of two or more small consecutive subdiagonal elements.

Input Parameters

s (local) INTEGER.
REAL for slamsh
DOUBLE PRECISION for dlamsh

	Array, `DIMENSION` (*lds*,*). On entry, the matrix of shifts. Only the 2x2 diagonal of *s* is referenced. It is assumed that *s* has *jblk* double shifts (size 2).
lds	(local) `INTEGER`. On entry, the leading dimension of *S*; unchanged on exit. $1 < nbulge \leq jblk \leq lds/2$.
nbulge	(local) `INTEGER`. On entry, the number of bulges to send through *h* (> 1). *nbulge* should be less than the maximum determined (*jblk*). $1 < nbulge \leq jblk \leq lds/2$.
jblk	(local) `INTEGER`. On entry, the leading dimension of *S*; unchanged on exit.
h	(local) `INTEGER`. `REAL` for `slamsh` `DOUBLE PRECISION` for `dlamsh` Array, `DIMENSION` (*lds*, *n*). On entry, the local matrix to apply the shifts on. *h* should be aligned so that the starting row is 2.
ldh	(local) `INTEGER`. On entry, the leading dimension of *H*; unchanged on exit.
n	(local) `INTEGER`. On entry, the size of *H*. If all the bulges are expected to go through, *n* should be at least 4*nbulge*+2. Otherwise, *nbulge* may be reduced by this routine.
ulp	(local) `REAL` for `slamsh` `DOUBLE PRECISION` for `dlamsh` On entry, machine precision. Unchanged on exit.

Output Parameters

s	On exit, the data is rearranged in the best order for applying.
nbulge	On exit, the maximum number of bulges that can be sent through.
h	On exit, the data is destroyed.

?laref

Applies Householder reflectors to matrices on either their rows or columns.

Syntax

```
call slaref (type, a, lda, wantz, z, ldz, block, irow1, icol1, istart, istop,
    itmp1, itm2, liloz, lihiz, vecs, v2, v3, t1, t2, t3)
call dlaref (type, a, lda, wantz, z, ldz, block, irow1, icol1, istart, istop,
    itmp1, itm2, liloz, lihiz, vecs, v2, v3, t1, t2, t3)
```

Description

This routine applies one or several Householder reflectors of size 3 to one or two matrices (if column is specified) on either their rows or columns.

Input Parameters

type (global) `CHRACTER*1.`
If *type* = `'R'`, apply reflectors to the rows of the matrix (apply from left). Otherwise, apply reflectors to the columns of the matrix. Unchanged on exit.

a (global)
`REAL` for `slaref`
`DOUBLE PRECISION` for `dlaref`
Array, `DIMENSION` (*lda*, `*`). On entry, the matrix to receive the reflections.

lda (local) `INTEGER`.
On entry, the leading dimension of *A*; unchanged on exit.

wantz (global) `LOGICAL`.
If *wantz* = `.TRUE.`, apply any column reflections to *Z* as well.
If *wantz* = `.FALSE.`, do no additional work on *Z*.

z (global)
`REAL` for `slaref`
`DOUBLE PRECISION` for `dlaref`
Array, `DIMENSION` (*ldz*, `*`). On enrty, the second matrix to receive column reflections.

ldz (local) `INTEGER`.
On entry, the leading dimension of *Z*; unchanged on exit.

block (global). LOGICAL.
= .TRUE. : apply several reflectors at once and read their data from the *vecs* array;
= .FALSE.: apply the single reflector given by *v2*, *v3*, *t1*, *t2*, and *t3*.

ipow1 (local) INTEGER.
On entry, the local row element of the matrix *A*.

icol1 (local) INTEGER.
On entry, the local column element of the matrix *A*.

istart (global) INTEGER.
Specifies the "number" of the first reflector.
istart is used as an index into *vecs* if *block* is set. *istart* is ignored if *block* is .FALSE..

istop (global) INTEGER.
Specifies the "number" of the last reflector.
istop is used as an index into *vecs* if *block* is set. *istop* is ignored if *block* is .FALSE..

itmp1 (local) INTEGER.
Starting range into *A*. For rows, this is the local first column. For columns, this is the local first row.

itmp2 (local) INTEGER.
Ending range into *A*. For rows, this is the local last column. For columns, this is the local last row.

liloz , *lihiz* (local). INTEGER.
Serve the same purpose as *itmp1*, *itmp2* but for *Z* when *wantz* is set.

vecs (global)
REAL for slaref
DOUBLE PRECISION for dlaref.
Array of size 3*n* (matrix size). This array holds the size 3 reflectors one after another and is only accessed when *block* is .TRUE.

v2,*v3*,*t1*,*t2*,*t3* (global). INTEGER.
REAL for slaref
DOUBLE PRECISION for dlaref.
These parameters hold information on a single size 3 Householder reflector and are read when *block* is .FALSE., and overwritten when *block* is .TRUE..

Output Parameters

a	On exit, the updated matrix.
z	Changed only if *wantz* is set. If *wantz* is .FALSE., *z* is not referenced.
ipow1	Undefined.
icol1	Undefined.
v2,*v3*,*t1*,*t2*,*t3*	These parameters are read when *block* is .FALSE., and overwritten when *block* is .TRUE..

?lasorte

Sorts eigenpairs by real and complex data types.

Syntax

```
call slasorte (s, lds, j, out, info)
call dlasorte (s, lds, j, out, info)
```

Description

This routine sorts eigenpairs so that real eigenpairs are together and complex eigenpairs are together. This helps to employ 2x2 shifts easily since every 2nd subdiagonal is guaranteed to be zero. This routine does no parallel work and makes no calls.

Input Parameters

s	(local) INTEGER. REAL for slasorte DOUBLE PRECISION for dlasorte Array, DIMENSION (*lds*). On entry, a matrix already in Schur form.
lds	(local) INTEGER. On entry, the leading dimension of the array *s*; unchanged on exit.
j	(local) INTEGER. On entry, the order of the matrix *S*; unchanged on exit.

out (local) INTEGER.
REAL for slasorte
DOUBLE PRECISION for dlasorte

Array, DIMENSION (*j*x2).
The work buffer required by the routine.

info (local) INTEGER.
Set, if the input matrix had an odd number of real eigenvalues and things could not be paired or if the input matrix *S* was not originally in Schur form.
0 indicates successful completion.

Output Parameters

s On exit, the diagonal blocks of *S* have been rewritten to pair the eigenvalues. The resulting matrix is no longer similar to the input.

out Work buffer.

?lasrt2

Sorts numbers in increasing or decreasing order.

Syntax

```
call slasrt2 (id, n, d, key, info)
call dlasrt2 (id, n, d, key, info)
```

Description

This routine is modified LAPACK routine ?lasrt, which sorts the numbers in *d* in increasing order (if *id* = 'I') or in decreasing order (if *id* = 'D'). It uses Quick Sort, reverting to Insertion Sort on arrays of size ≤ 20. Dimension of stack limits *n* to about 2^{32}.

Input Parameters

id CHARACTER*1.
= 'I': sort *d* in increasing order;
= 'D': sort *d* in decreasing order.

n INTEGER. The length of the array *d*.

d REAL for slasrt2
DOUBLE PRECISION for dlasrt2.
Array, DIMENSION (*n*).
On entry, the array to be sorted.

key INTEGER.
Array, DIMENSION (*n*).
On entry, *key* contains a key to each of the entries in *d*().
Typically, *key*(*i*) = *i* for all *i* .

Output Parameters

d On exit, *d* has been sorted into increasing order ($d(1) \le \ldots \le d(n)$) or into decreasing order ($d(1) \ge \ldots \ge d(n)$), depending on *id*.

info INTEGER.
= 0: successful exit
< 0: if *info* = -i, the *i*-th argument had an illegal value.

key On exit, *key* is permuted in exactly the same manner as *d*() was permuted from input to output. Therefore, if *key*(*i*) = *i* for all *i* upon input, then * *d*_out(*i*) = *d*_in(*key*(*i*)).

?stein2

Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix, using inverse iteration.

Syntax

```
call sstein2 (n, d, e, m, w, iblock, isplit, orfac, z, ldz,
   work, iwork, ifail, info)
call dstein2 (n, d, e, m, w, iblock, isplit, orfac, z, ldz,
   work, iwork, ifail, info)
```

Description

This routine is a modified LAPACK routine ?stein. It computes the eigenvectors of a real symmetric tridiagonal matrix *T* corresponding to specified eigenvalues, using inverse iteration.

The maximum number of iterations allowed for each eigenvector is specified by an internal parameter *maxits* (currently set to 5).

Input Parameters

n — INTEGER. The order of the matrix *T* ($n \ge 0$).

m — INTEGER. The number of eigenvectors to be found ($0 \le m \le n$).

d, e, w — REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Arrays:
d(*), DIMENSION (*n*).
The *n* diagonal elements of the tridiagonal matrix *T*.

e(*), DIMENSION (*n*).
The (*n*-1) subdiagonal elements of the tridiagonal matrix *T*, in elements 1 to *n*-1. *e*(*n*) need not be set.

w(*), DIMENSION (*n*).
The first *m* elements of *w* contain the eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block. (The output array *w* from ?stebz with ORDER = 'B' is expected here).

The dimension of *w* must be at least max(1, *n*).

iblock — INTEGER.
Array, DIMENSION (*n*).
The submatrix indices associated with the corresponding eigenvalues in *w* ;
iblock(*i*) = 1, if eigenvalue *w*(*i*) belongs to the first submatrix from the top,
iblock(*i*) = 2, if eigenvalue *w*(*i*) belongs to the second submatrix, etc. (The output array *iblock* from ?stebz is expected here).

isplit — INTEGER.
Array, DIMENSION (*n*).
The splitting points, at which *T* breaks up into submatrices. The first submatrix consists of rows/columns 1 to *isplit*(1), the second submatrix consists of rows/columns *isplit*(1)+1 through *isplit*(2), etc. (The output array *isplit* from ?stebz is expected here).

orfac — REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
orfac specifies which eigenvectors should be orthogonalized. Eigenvectors that correspond to eigenvalues which are within *orfac**$\|T\|$ of each other are to be orthogonalized.

ldz — INTEGER. The leading dimension of the output array *z*; *ldz* ≥ max(1, *n*).

work — REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Workspace array, DIMENSION (5*n*).

iwork — INTEGER.
Workspace array, DIMENSION (*n*).

Output Parameters

z — REAL for sstein2
DOUBLE PRECISION for dstein2
Array, DIMENSION (*ldz*, *m*).
The computed eigenvectors. The eigenvector associated with the eigenvalue *w*(*i*) is stored in the *i*-th column of *z*. Any vector that fails to converge is set to its current iterate after *maxits* iterations.

ifail — INTEGER. Array, DIMENSION (*m*).
On normal exit, all elements of *ifail* are zero. If one or more eigenvectors fail to converge after *maxits* iterations, then their indices are stored in the array *ifail*.

info — INTEGER.
info = 0, the exit is successful.
info < 0: if *info* = -*i*, the *i*-th had an illegal value.
info > 0: if *info* = *i*, then *i* eigenvectors failed to converge in *maxits* iterations. Their indices are stored in the array *ifail*.

?dbtf2

Computes an LU factorization of a general band matrix with no pivoting (local unblocked algorithm).

Syntax

```
call sdbtf2 (m, n, kl, ku, ab, ldab, info)
call ddbtf2 (m, n, kl, ku, ab, ldab, info)
call cdbtf2 (m, n, kl, ku, ab, ldab, info)
call zdbtf2 (m, n, kl, ku, ab, ldab, info)
```

Description

This routine computes an *LU* factorization of a general real/complex *m*-by-*n* band matrix *A* without using partial pivoting with row interchanges.

This is the unblocked version of the algorithm, calling BLAS Level 2 Routines.

Input Parameters

m — INTEGER. The number of rows of the matrix *A* ($m \geq 0$).

n — INTEGER. The number of columns in *A* ($n \geq 0$).

kl — INTEGER. The number of sub-diagonals within the band of *A* ($kl \geq 0$).

ku — INTEGER. The number of super-diagonals within the band of *A* ($ku \geq 0$).

ab — REAL for sdbtf2
DOUBLE PRECISION for ddbtf2
COMPLEX for cdbtf2
COMPLEX*16 for zdbtf2.
Array, DIMENSION (*ldab*, *n*).
The matrix *A* in band storage, in rows $kl+1$ to $2kl+ku+1$; rows 1 to *kl* of the array need not be set. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows: $ab(kl+ku+1+i-j,j) = A(i,j)$ for $\max(1,j-ku) \leq i \leq \min(m,j+kl)$.

ldab — INTEGER. The leading dimension of the array *ab*.
($ldab \geq 2kl + ku + 1$)

Output Parameters

ab On exit, details of the factorization: *U* is stored as an upper triangular band matrix with *kl*+*ku* superdiagonals in rows 1 to *kl*+*ku*+1, and the multipliers used during the factorization are stored in rows *kl*+*ku*+2 to 2**kl*+*ku*+1. See the *Application Notes* below for further details.

info INTEGER.
= 0: successful exit
< 0: if *info* = -*i*, the *i*-th argument had an illegal value,
> 0: if *info* = +*i*, *u*(*i*,*i*) is 0. The factorization has been completed, but the factor *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

Application Notes

The band storage scheme is illustrated by the following example, when $m = n = 6$, $kl = 2$, $ku = 1$:

on entry

$$\begin{bmatrix} * & a12 & a23 & a34 & a45 & a56 \\ a11 & a22 & a33 & a44 & a55 & a66 \\ a21 & a32 & a43 & a54 & a65 & * \\ a31 & a42 & a53 & a64 & * & * \end{bmatrix}$$

on exit

$$\begin{bmatrix} * & u12 & u23 & u34 & u45 & u56 \\ u11 & u22 & u33 & u44 & u55 & u66 \\ m21 & m32 & m43 & m54 & m65 & * \\ m31 & m42 & m53 & m64 & * & * \end{bmatrix}$$

The routine does not use array elements marked *; elements marked + need not be set on entry, but the routine requires them to store elements of *U*, because of fill-in resulting from the row interchanges.

?dbtrf

Computes an LU factorization of a general band matrix with no pivoting (local blocked algorithm).

Syntax

```
call sdbtrf (m, n, kl, ku, ab, ldab, info)
call ddbtrf (m, n, kl, ku, ab, ldab, info)
call cdbtrf (m, n, kl, ku, ab, ldab, info)
call zdbtrf (m, n, kl, ku, ab, ldab, info)
```

Description

This routine computes an LU factorization of a real *m*-by-*n* band matrix *A* without using partial pivoting or row interchanges.

This is the blocked version of the algorithm, calling BLAS Level 3 Routines.

Input Parameters

m — `INTEGER`. The number of rows of the matrix *A* ($m \ge 0$).

n — `INTEGER`. The number of columns in *A* ($n \ge 0$).

kl — `INTEGER`. The number of sub-diagonals within the band of *A* ($kl \ge 0$).

ku — `INTEGER`. The number of super-diagonals within the band of *A* ($ku \ge 0$).

ab — `REAL` for `sdbtrf`
`DOUBLE PRECISION` for `ddbtrf`
`COMPLEX` for `cdbtrf`
`COMPLEX*16` for `zdbtrf`.
Array, `DIMENSION` (*ldab*, *n*).
The matrix *A* in band storage, in rows $kl+1$ to $2kl+ku+1$; rows 1 to *kl* of the array need not be set. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows: $ab(kl+ku+1+i-j,j) = A(i,j)$ for $\max(1,j-ku) \le i \le \min(m,j+kl)$.

ldab — `INTEGER`. The leading dimension of the array *ab*.
($ldab \ge 2kl + ku +1$)

Output Parameters

ab On exit, details of the factorization: *U* is stored as an upper triangular band matrix with *kl*+*ku* superdiagonals in rows 1 to *kl*+*ku*+1, and the multipliers used during the factorization are stored in rows *kl*+*ku*+2 to 2**kl*+*ku*+1. See the *Application Notes* below for further details.

info INTEGER.
= 0: successful exit
< 0: if *info* = -*i*, the *i*-th argument had an illegal value,
> 0: if *info* = +*i*, *u*(*i*,*i*) is 0. The factorization has been completed, but the factor *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

Application Notes

The band storage scheme is illustrated by the following example, when *m* = *n* = 6, *kl* = 2, *ku* = 1:

on entry on exit

$$\begin{bmatrix} * & a12 & a23 & a34 & a45 & a56 \\ a11 & a22 & a33 & a44 & a55 & a66 \\ a21 & a32 & a43 & a54 & a65 & * \\ a31 & a42 & a53 & a64 & * & * \end{bmatrix} \qquad \begin{bmatrix} * & u12 & u23 & u34 & u45 & u56 \\ u11 & u22 & u33 & u44 & u55 & u66 \\ m21 & m32 & m43 & m54 & m65 & * \\ m31 & m42 & m53 & m64 & * & * \end{bmatrix}$$

The routine does not use array elements marked *.

?dttrf

Computes an LU factorization of a general tridiagonal matrix with no pivoting (local blocked algorithm).

Syntax

```
call sdttrf (n, dl, d, du, info)
call ddttrf (n, dl, d, du, info)
call cdttrf (n, dl, d, du, info)
call zdttrf (n, dl, d, du, info)
```

Description

This routine computes an *LU* factorization of a real or complex tridiagonal matrix *A* using elimination without partial pivoting.

The factorization has the form $A = LU$, where *L* is a product of unit lower bidiagonal matrices and *U* is upper triangular with nonzeros only in the main diagonal and first superdiagonal.

Input Parameters

n	`INTEGER`. The order of the matrix *A* ($n \geq 0$).
dl, d, du	`REAL` for `sdttrf` `DOUBLE PRECISION` for `ddttrf` `COMPLEX` for `cdttrf` `COMPLEX*16` for `zdttrf`. Arrays containing elements of *A*. The array *dl* of `DIMENSION` $(n - 1)$ contains the sub-diagonal elements of *A*. The array *d* of `DIMENSION` *n* contains the diagonal elements of *A*. The array *du* of `DIMENSION` $(n - 1)$ contains the super-diagonal elements of *A*.

Output Parameters

dl	Overwritten by the $(n-1)$ multipliers that define the matrix *L* from the *LU* factorization of *A*.
d	Overwritten by the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.
du	Overwritten by the $(n-1)$ elements of the first super-diagonal of *U*.
info	`INTEGER`. = 0: successful exit < 0: if *info* = - *i*, the *i*-th argument had an illegal value, > 0: if *info* = *i*, $u(i,i)$ is exactly 0. The factorization has been completed, but the factor *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

?dttrsv

Solves a general tridiagonal system of linear equations using the LU factorization computed by `?dttrf`.

Syntax

```
call sdttrsv (uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
call ddttrsv (uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
call cdttrsv (uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
call zdttrsv (uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
```

Description

This routine solves one of the following systems of linear equations:

$LX = B$, $L^TX = B$, or $L^HX = B$,

$UX = B$, $U^TX = B$, or $U^HX = B$

with factors of the tridiagonal matrix A from the LU factorization computed by `?dttrf`.

Input Parameters

uplo — `CHARACTER*1`.
Specifies whether to solve with L or U.

trans — `CHARACTER`. Must be `'N'` or `'T'` or `'C'`.
Indicates the form of the equations:
If *trans* = `'N'`, then $AX = B$ is solved for X (no transpose).
If *trans* = `'T'`, then $A^TX = B$ is solved for X (transpose).
If *trans* = `'C'`, then $A^HX = B$ is solved for X (conjugate transpose).

n — `INTEGER`. The order of the matrix A ($n \geq 0$).

nrhs — `INTEGER`. The number of right-hand sides, i.e., the number of columns in the matrix B ($nrhs \geq 0$).

dl,d,du,b — `REAL` for `sdttrsv`
`DOUBLE PRECISION` for `ddttrsv`
`COMPLEX` for `cdttrsv`
`COMPLEX*16` for `zdttrsv`.
Arrays of `DIMENSION`s: *dl*(*n* - 1), *d*(*n*), *du*(*n* - 1), *b*(*ldb*, *nrhs*).

The array `dl` contains the (n - 1) multipliers that define the matrix L from the LU factorization of A.
The array `d` contains `n` diagonal elements of the upper triangular matrix U from the LU factorization of A.
The array `du` contains the (n - 1) elements of the first super-diagonal of U.
On entry, the array `b` contains the right-hand side matrix B.

ldb — `INTEGER`. The leading dimension of the array *b*; $ldb \geq \max(1, n)$.

Output Parameters

b — Overwritten by the solution matrix X.

info — `INTEGER`. If *info*=0, the execution is successful.
If *info* = *-i*, the *i*-th parameter had an illegal value.

?pttrsv

Solves a symmetric (Hermitian) positive-definite tridiagonal system of linear equations, using the LDL^H factorization computed by `?pttrf`.

Syntax

```
call spttrsv (trans, n, nrhs, d, e, b, ldb, info)
call dpttrsv (trans, n, nrhs, d, e, b, ldb, info)
call cpttrsv (uplo, trans, n, nrhs, d, e, b, ldb, info)
call zpttrsv (uplo, trans, n, nrhs, d, e, b, ldb, info)
```

Description

This routine solves one of the triangular systems:

$$L^T X = B, \text{ or } LX = B \quad \text{for real flavors,}$$

or

$$LX = B, \text{ or } L^H X = B,$$
$$UX = B, \text{ or } U^H X = B \quad \text{for complex flavors,}$$

where L (or U for complex flavors) is the Cholesky factor of a Hermitian positive-definite tridiagonal matrix A such that

$A = LDL^H$ (computed by spttrf/dpttrf)

or

$A = U^H DU$ or $A = LDL^H$ (computed by cpttrf/zpttrf).

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix A is stored and the form of the factorization:

If *uplo* = 'U', *e* is the superdiagonal of U, and $A = U'DU$;
If *uplo* = 'L', *e* is the subdiagonal of L, and $A = LDL'$.

The two forms are equivalent, if A is real.

trans CHARACTER.
Specifies the form of the system of equations:

for real flavors:
if *trans* = 'N': $LX = B$ (no transpose)
if *trans* = 'T': $L^T X = B$ (transpose)

for complex flavors:
if *trans* = 'N': $LX = B$ (no transpose)
if *trans* = 'N': $LX = B$ (no transpose)
if *trans* = 'C': $U^H X = B$ (conjugate transpose)
if *trans* = 'C': $L^H X = B$ (conjugate transpose)

n INTEGER. The order of the tridiagonal matrix A. $n \geq 0$.

nrhs INTEGER. The number of right hand sides, that is, the number of columns of the matrix B. $nrhs \geq 0$.

d REAL array, DIMENSION (*n*). The *n* diagonal elements of the diagonal matrix D from the factorization computed by ?pttrf.

e COMPLEX array, DIMENSION (*n*-1). The (*n*-1) off-diagonal elements of the unit bidiagonal factor U or L from the factorization computed by ?pttrf. See *uplo*.

b COMPLEX array, DIMENSION (*ldb*, *nrhs*).
On entry, the right hand side matrix B.

ldb INTEGER.
The leading dimension of the array *b*.
ldb ≥ max(1, *n*).

Output Parameters

b On exit, the solution matrix *X*.

info INTEGER.
= 0: successful exit
< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?steqr2

Computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method.

Syntax

```
call ssteqr2 (compz, n, d, e, z, ldz, nr, work, info)
call dsteqr2 (compz, n, d, e, z, ldz, nr, work, info)
```

Description

This routine is a modified version of LAPACK routine ?steqr. The routine ?steqr2 computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method. ?steqr2 is modified from ?steqr to allow each ScaLAPACK process running ?steqr2 to perform updates on a distributed matrix Q. Proper usage of ?steqr2 can be gleaned from examination of ScaLAPACK routine p?syev.

Input Parameters

compz CHARACTER*1. Must be 'N' or 'I'.

If *compz* = 'N', the routine computes eigenvalues only.
If *compz* = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix *T*.
z must be initialized to the identity matrix by p?laset or ?laset prior to entering this subroutine.

n INTEGER. The order of the matrix T ($n \geq 0$).

d, e, work REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Arrays:
d contains the diagonal elements of *T*.
The dimension of *d* must be at least max(1, *n*).

e contains the (*n*-1) subdiagonal elements of *T*.
The dimension of *e* must be at least max(1, *n*-1).

work is a workspace array.
The dimension of *work* is max(1, 2**n*-2).
If *compz* = 'N', then *work* is not referenced.

z (local)
REAL for ssteqr2
DOUBLE PRECISION for dsteqr2
Array, global DIMENSION (*n*, *n*), local DIMENSION (*ldz*, *nr*).
If *compz* = 'V', then *z* contains the orthogonal matrix used in the reduction to tridiagonal form.

ldz INTEGER. The leading dimension of the array *z*. Constrains:
$ldz \geq 1$,
$ldz \geq$ max(1, *n*), if eigenvectors are desired.

nr INTEGER. *nr* = max(1, *numroc*(*n*, *nb*, *myprow*, 0, *nprocs*)).
If *compz* = 'N', then *nr* is not referenced.

Output Parameters

d REAL array, DIMENSION (*n*), for ssteqr2.
DOUBLE PRECISION array, DIMENSION (*n*), for dsteqr2.
On exit, the eigenvalues in ascending order, if *info* = 0.
See also *info*.

e REAL array, DIMENSION (*n*-1), for ssteqr2.
DOUBLE PRECISION array, DIMENSION (*n*-1), for dsteqr2.
On exit, *e* has been destroyed.

z (local)
REAL for ssteqr2
DOUBLE PRECISION for dsteqr2
Array, global DIMENSION (*n*, *n*), local DIMENSION (*ldz*, *nr*).
On exit, if *info* = 0, then, if *compz* = 'V', *z* contains the orthonormal eigenvectors

of the original symmetric matrix, and if *compz* = 'I', *z* contains the orthonormal eigenvectors of the symmetric tridiagonal matrix.
If *compz* = 'N', then *z* is not referenced.

info INTEGER.
info = 0, the exit is successful.
info < 0: if *info* = -*i*, the *i*-th had an illegal value.
info > 0: the algorithm has failed to find all the eigenvalues in a total of 30*n* iterations; if *info* = *i*, then *i* elements of *e* have not converged to zero; on exit, *d* and *e* contain the elements of a symmetric tridiagonal matrix, which is orthogonally similar to the original matrix.

Utility Functions and Routines

This section describes ScaLAPACK utility functions and routines. Summary information about these routines is given in the following table:

Table 7-2 ScaLAPACK Utility Functions and Routines

Routine Name	Data Types	Description
p?labad	s,d	Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.
p?lachkieee	s,d	Performs a simple check for the features of the IEEE standard. (C interface function).
p?lamch	s,d	Determines machine parameters for floating-point arithmetic.
p?lasnbt	s,d	Computes the position of the sign bit of a floating-point number. (C interface function).
pxerbla		Error handling routine called by ScaLAPACK routines.

p?labad

Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.

Syntax

```
call pslabad (ictxt, small, large)
call pdlabad (ictxt, small, large)
```

Description

This routine takes as input the values computed by `p?lamch` for underflow and overflow, and returns the square root of each of these values if the log of *`large`* is sufficiently large. This subroutine is intended to identify machines with a large exponent range, such as the Crays, and redefine the underflow and overflow limits to be the square roots of the values computed by `p?lamch`. This subroutine is needed because `p?lamch` does not compensate for poor arithmetic in the upper half of the exponent range, as is found on a Cray.

In addition, this routine performs a global minimization and maximization on these values, to support heterogeneous computing networks.

Input Parameters

`ictxt` (global) `INTEGER.`
The BLACS context handle in which the computation takes place.

`small` (local).
`REAL PRECISION` for `pslabad.`
`DOUBLE PRECISION` for `pdlabad.`
On entry, the underflow threshold as computed by `p?lamch.`

`large` (local).
`REAL PRECISION` for `pslabad.`
`DOUBLE PRECISION` for `pdlabad.`
On entry, the overflow threshold as computed by `p?lamch`.

Output Parameters

`small` (local).
On exit, if *`log10(large)`* is sufficiently large, the square root of *`small`*, otherwise unchanged.

large (local).
On exit, if *log10(large)* is sufficiently large, the square root of *large*, otherwise unchanged.

p?lachkieee

Performs a simple check for the features of the IEEE standard. (C interface function).

Syntax

```
void pslachkieee (int *isieee, float *rmax, float *rmin);
void pdlachkieee (int *isieee, float *rmax, float *rmin);
```

Description

This routine performs a simple check to make sure that the features of the IEEE standard are implemented. In some implementations, `p?lachkieee` may not return.

Note that all arguments are call-by-reference so that this routine can be directly called from Fortran code.

This is a ScaLAPACK internal subroutine and arguments are not checked for unreasonable values.

Input Parameters

rmax (local).
`REAL` for `pslachkieee`
`DOUBLE PRECISION` for `pdlachkieee`
The overflow threshold (= `?lamch ('O')`).

rmin (local).
`REAL` for `pslachkieee`
`DOUBLE PRECISION` for `pdlachkieee`
The underflow threshold (= `?lamch ('U')`).

Output Parameters

isieee (local).INTEGER.
On exit, *isieee* = 1 implies that all the features of the IEEE standard that we rely on are implemented.
On exit, *isieee* = 0 implies that some the features of the IEEE standard that we rely on are missing.

p?lamch

Determines machine parameters for floating-point arithmetic.

Syntax

```
val = pslamch (ictxt, cmach)
val = pdlamch (ictxt, cmach)
```

Description

This function determines single precision machine parameters.

Input Parameters.

ictxt (global). INTEGER. The BLACS context handle in which the computation takes place.

cmach (global) CHARACTER*1.
Specifies the value to be returned by p?lamch:
= 'E' or 'e', p?lamch := eps
= 'S' or 's , p?lamch := sfmin
= 'B' or 'b', p?lamch := base
= 'P' or 'p', p?lamch := eps*base
= 'N' or 'n', p?lamch := t
= 'R' or 'r', p?lamch := rnd
= 'M' or 'm', p?lamch := emin
= 'U' or 'u', p?lamch := rmin
= 'L' or 'l', p?lamch := emax
= 'O' or 'o', p?lamch := rmax

where

`eps` = relative machine precision
`sfmin` = safe minimum, such that 1/sfmin does not overflow
`base` = base of the machine
`prec` = `eps*base`
`t` = number of (base) digits in the mantissa
`rnd` = 1.0 when rounding occurs in addition, 0.0 otherwise
`emin` = minimum exponent before (gradual) underflow
`rmin` = underflow threshold - $base^{(emin-1)}$
`emax` = largest exponent before overflow
`rmax` = overflow threshold - $(base^{emax})*(1-eps)$

Output Parameter

val the value returned by the fuction.

p?lasnbt

Computes the position of the sign bit of a floating-point number. (C interface function).

Syntax

```
void pslasnbt (int *ieflag);
void pdlasnbt (int *ieflag);
```

Description

This routine finds the position of the signbit of a single/double precision floating point number. This routine assumes IEEE arithmetic, and hence, tests only the 32^{nd} bit (for single precision) or 32^{nd} and 64^{th} bits (for double precision) as a possibility for the signbit. `sizeof(int)` is assumed equal to 4 bytes.

If a compile time flag (`NO_IEEE`) indicates that the machine does not have IEEE arithmetic, *ieflag* = 0 is returned.

Output Parameters

ieflag INTEGER.
This flag indicates the position of the signbit of any single/double precision floating point number.

ieflag = 0, if the compile time flag `NO_IEEE` indicates that the machine does not have IEEE arithmetic, or if `sizeof(int)` is different from 4 bytes.
ieflag = 1 indicates that the signbit is the 32nd bit for a single precision routine.
In the case of a double precision routine:
ieflag = 1 indicates that the signbit is the 32nd bit (Big Endian).
ieflag = 2 indicates that the signbit is the 64th bit (Little Endian).

pxerbla

Error handling routine called by ScaLAPACK routines.

Syntax

`call pxerbla (`*`ictxt, srname, info`*`)`

Description

This routine is an error handler for the ScaLAPACK routines. It is called by a ScaLAPACK routine if an input parameter has an invalid value.
A message is printed. Program execution is not terminated. For the ScaLAPACK driver and computational routines, a `RETURN` statement is issued following the call to `pxerbla`. Control returns to the higher-level calling routine, and it is left to the user to determine how the program should proceed. However, in the specialized low-level ScaLAPACK routines (auxiliary routines that are Level 2 equivalents of computational routines), the call to `pxerbla()` is immediately followed by a call to `BLACS_ABORT()` to terminate program execution since recovery from an error at this level in the computation is not possible.

It is always good practice to check for a nonzero value of *info* on return from a ScaLAPACK routine.
Installers may consider modifying this routine in order to call system-specific exception-handling facilities.

Input Parameters

ictxt (global) `INTEGER`
The BLACS context handle, indicating the global context of the operation. The context itself is global.

srname (global) `CHARACTER*6`
The name of the routine which called `pxerbla`.

info (global) INTEGER.
The position of the invalid parameter in the parameter list of the calling routine.

Sparse Solver Routines

8

Intel® Math Kernel Library (Intel® MKL) provides a user-callable direct sparse solver software to solve symmetric and symmetrically-structured matrices with real or complex coefficients. For sparse symmetric matrices, this solver can solve both positive definite and indefinite systems.

The terms and concepts required to understand the use of the Intel MKL direct sparse solver subroutines are discussed in the Linear Solvers Basics appendix. If you are familiar with direct sparse solvers and sparse matrix storage schemes, you can omit reading these sections and go directly to the interface descriptions. The direct sparse solver PARDISO* is described in the section that follows. After that, an alternative interface (referred to here as DSS interface) that consists of several Intel MKL routines implementing the step-by-step solution process is described.

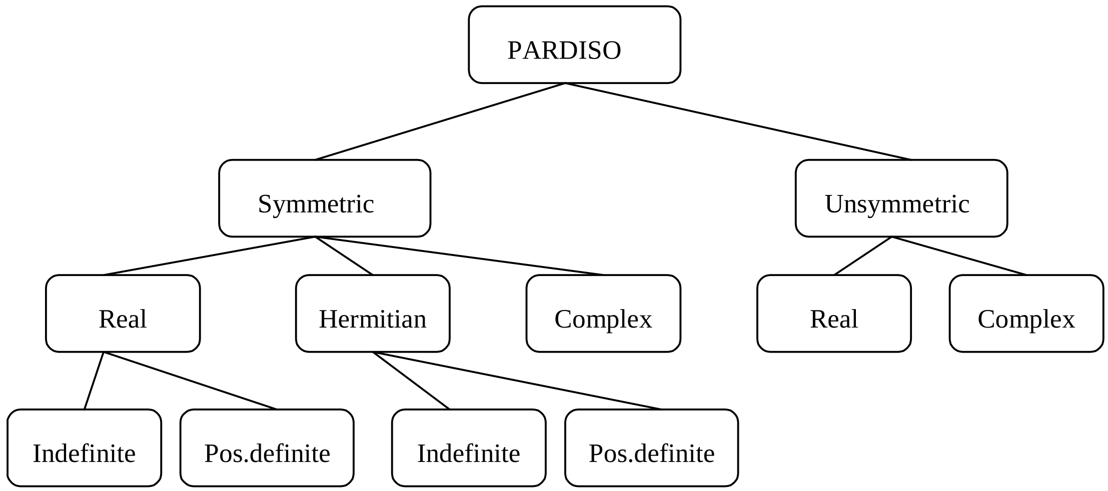
PARDISO - Parallel Direct Sparse Solver Interface

This section describes the interface to the shared-memory multiprocessing parallel direct sparse solver known as PARDISO. The interface is Fortran, but can be called from C programs by observing Fortran parameter passing and naming conventions used by the supported compilers and operating systems. A discussion of the algorithms used in PARDISO and more information on the solver can be found at http://www.computational.unibas.ch/cs/scicomp.

The PARDISO package is a high-performance, robust, memory efficient and easy to use software for solving large sparse symmetric and unsymmetric linear systems of equations on shared memory multiprocessors. The solver uses a combination of left- and right-looking Level-3 BLAS supernode techniques [10]. In order to improve sequential and parallel sparse numerical factorization performance, the algorithms are based on a Level-3 BLAS update and pipelining parallelism is exploited with a combination of left- and right-looking supernode techniques [6, 7, 8, 9]. The parallel pivoting methods allow complete supernode pivoting in order to compromise numerical stability and scalability during the factorization process. For sufficiently large problem sizes, numerical experiments demonstrate that the scalability of the parallel algorithm is nearly independent of the shared-memory multiprocessing architecture and a speedup of up to seven using eight processors has been observed.

PARDISO supports, as illustrated in Figure 1, a wide range of sparse matrix types and computes the solution of real or complex, symmetric, structurally symmetric or unsymmetric, positive definite, indefinite or Hermitian sparse linear system of equations on shared-memory multiprocessing architectures.

Figure 8-1 Sparse Matrices That Can be Solved With PARDISO



You can find example code that uses PARDISO interface routine to solve systems of linear equations in PARDISO Code Examples section in the appendix.

pardiso

Calculates the solution of a set of sparse linear equations with multiple right-hand sides.

Syntax

Fortran:

```
call pardiso(pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
    perm, nrhs, iparm, msglvl, b, x, error)
```

C:

```
pardiso(pt, &maxfct, &mnum, &mtype, &phase, &n, a, ia, ja, perm, &nrhs,
    iparm, &msglvl, b, x, &error);
```

(An underscore may or may not be required after "`pardiso`" depending on the OS and compiler conventions for that OS).

Interface:

```
SUBROUTINE pardiso(pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
                   perm, nrhs, iparm, msglvl, b, x, error)
INTEGER*4 pt(64)
INTEGER*4 maxfct, mnum, mtype, phase, n, nrhs, error,
          ia(*), ja(*), perm(*), iparm(*)
REAL*8 a(*), b(n,nrhs), x(n,nrhs)
```

Note that the above interface is given for the 32-bit architectures. For 64-bit architectures, the argument `pt(64)` must be defined as `INTEGER*8` type.

Description

PARDISO calculates the solution of a set of sparse linear equations

$$AX = B$$

with multiple right-hand sides, using a parallel *LU, LDL* or LL^T factorization, where A is an n-by-n matrix and X and B are n-by-nrhs matrices. PARDISO performs the following analysis steps depending on the structure of the input matrix A.

Symmetric Matrices: The solver first computes a symmetric fill-in reducing permutation P based on either the minimum degree algorithm [5] or the nested dissection algorithm from the METIS package [2] (included with Intel MKL), followed by the parallel left-right looking numerical Cholesky factorization [10] of $PAP^T = LL^T$ for symmetric positive-definite matrices or $PAP^T = LDL^T$ for symmetric, indefinite matrices. The solver uses no pivoting in these steps and an approximation of X is found by forward and backward substitution and iterative refinements.

Structurally Symmetric Matrices: The solver first computes a symmetric fill-in reducing permutation P followed by the parallel numerical factorization of $PAP^T = QLU^T$. The solver uses partial pivoting in the supernodes and an approximation of X is found by forward and backward substitution and iterative refinements.

Unsymmetric Matrices: The solver first computes a non-symmetric permutation P_{MPS} and scaling matrices D_r and D_c with the aim to place large entries on the diagonal which enhances greatly the reliability of the numerical factorization process [1]. In the next step the solver computes a fill-in reducing permutation P based on the matrix $P_{MPS}A + (P_{MPS}A)^T$ followed by the parallel numerical factorization

$$QLUR = PP_{MPS}D_rAD_cP$$

with supernode pivoting matrices Q and R. When the factorization algorithm reaches a point where it cannot factorize the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [4]. The magnitude of the potential pivot is tested against a constant threshold of $\varepsilon = \alpha \cdot \|A_2\|_\infty$, where ε is the machine precision $A_2 = PP_{MPS}D_rAD_c$, and $\|A_2\|_\infty$ is the infinity norm of the scaled and permuted matrix A. Therefore any tiny pivots encountered during elimination are set to the $\mathrm{sign}(l_{ii}) \cdot \varepsilon \cdot \|A_2\|_\infty$ — this trades off some numerical stability for the ability to keep pivots from getting too small. Although many failures could render the factorization well-defined but essentially useless, in practice it is observed that the diagonal elements are rarely modified for a large class of matrices. The result of this pivoting approach is that the factorization is, in general, not exact and iterative refinement may be needed.

Direct-Iterative Preconditioning for Unsymmetric Linear Systems. The solver also allows a combination of direct and iterative methods [11] in order to accelerate the linear solution process for transient simulation. A majority of applications of sparse solvers require solutions of systems with gradually changing values of the nonzero coefficient matrix, but the same identical sparsity pattern. In these applications, the analysis phase of the solvers has to be performed only once and the numerical factorizations are the important time-consuming steps during the simulation. PARDISO uses a numerical factorization $A = LU$ for the first system and applies these exact factors L and U for the next steps in a preconditioned Krylow-Subspace iteration. If the iteration does not converge, the solver will automatically switch back to the numerical factorization. This method can be applied for unsymmetric matrices in PARDISO and the user can select the method using only one input parameter. For further details see the parameter description (`iparm(4)`, `iparm(20)`).

The sparse data storage in PARDISO follows the scheme described in [Sparse Matrix Storage Format] section with `ja` standing for `columns`, `ia` for `rowIndex`, and `a` for `values`.
The algorithms in PARDISO require column indices `ja` to be increasingly ordered per row and the presence of the diagonal element per row for any symmetric or structurally symmetric matrix. The unsymmetric matrices need no diagonal elements in the PARDISO solver.

There are four tasks that PARDISO is capable of performing, namely analysis and symbolic factorization, numerical factorization, forward and backward substitution including iterative refinement and finally the termination to release all internal solver memory. When an input data structure is not accessed in a call, a `NULL` pointer or any valid address can be passed as a place holder for that argument.

Input Parameters

pt — `INTEGER*4` for 32-bit operating systems
`INTEGER*8` for 64-bit operating systems.
Array, `DIMENSION` (64).
On entry, this is the solver internal data address pointer. Theses addresses are passed to the solver and all related internal memory management is organized through this pointer.

NOTE. *pt* is an integer array with 64 entries. It is very important that the pointer is initialized with zero at the first call of PARDISO. After that first call you should never modify the pointer, as a serious memory leak can occur. The integer length should be 4-byte on 32-bit operating systems and 8-byte on 64-bit operating systems.

maxfct — `INTEGER`.
Maximal number of factors with identical nonzero sparsity structure that the user would like to keep at the same time in memory. It is possible to store several different factorizations with the same nonzero structure at the same time in the internal data management of the solver. In most of the applications this value is equal to 1.
Note that PARDISO can process several matrices with identical matrix sparsity pattern and is able to store the factors of these matrices at the same time. Matrices with different sparsity structure can be kept in memory with different memory address pointers *pt*.

mnum `INTEGER`.
Actual matrix for the solution phase. With this scalar you can define the matrix that you would like to factorize. The value must be: $1 \leq$ *mnum* $\leq$ *maxfct*.
In most of the applications this value is equal to 1.

mtype `INTEGER`.
This scalar value defines the matrix type. The solver PARDISO supports the following matrices:

mtype = 1	real and structurally symmetric matrix
= 2	real and symmetric positive definite matrix
= -2	real and symmetric indefinite matrix
= 3	complex and structurally symmetric matrix
= 4	complex and Hermitian positive definite matrix
= -4	complex and Hermitian indefinite matrix
= 6	complex and symmetric matrix
= 11	real and unsymmetric matrix
= 13	complex and unsymmetric matrix

Note that this parameter influences the pivoting method.

phase `INTEGER`.
Controls the execution of the solver. It is a two-digit integer $i\,j$ ($10i + j$, $1 \leq i \leq 3$, $i < j \leq 3$ for normal execution modes). The i digit indicates the starting phase of execution, and j indicates the ending phase. PARDISO has the following phases of execution:

- Phase 1: Fill-reduction analysis and symbolic factorization
- Phase 2: Numerical factorization
- Phase 3: Forward and Backward solve including iterative refinements
- Termination and Memory Release Phase (*phase* ≤ 0)

If a previous call to the routine has computed information from previous phases, execution may start at any phase. The *phase* parameter can have the following values:

phase	**Solver Execution Steps**
11	Analysis, symbolic factorization
12	Analysis, symbolic factorization, numerical factorization
13	Analysis, symbolic factorization, numerical factorization, solve
22	Numerical factorization
23	Numerical factorization, solve

phase	**Solver Execution Steps**
33	Solve
0	Release internal memory for *L* and *U* matrix number `mnum`
-1	Release all internal memory for all matrices

n `INTEGER`.
Number of equations. This is the number of equations in the sparse linear systems of equations $A X = B$. Constraint: *n* > 0.

a `REAL/COMPLEX`
Array. Contains the nonzero values of the coefficient matrix *A* corresponding to the indices in `ja`. The size of *a* is the same as that of `ja` and the coefficient matrix can be either real or complex. The matrix must be stored in compressed sparse row format with increasing values of `ja` for each row. Refer to `values` array description in Sparse Matrix Storage Format for more details.

NOTE. The nonzeros of each row of the matrix *A* must be stored in increasing order. For symmetric or structural symmetric matrices it is also important that the diagonal elements are also available and stored in the matrix. If the matrix is symmetric, then the array *a* is only accessed in the factorization phase, in the triangular solution and iterative refinement phase. Unsymmetric matrices are accessed in all phases of the solution process.

`ia` `INTEGER`.
Array, dimension (*n*+1). For `i`≤ *n*, `ia`(`i`) points to the first column index of row `i` in the array `ja` in compressed sparse row format. That is, `ia`(`i`) gives the index of the element in array *a* that contains the first non-zero element from row `i` of *A*. The last element `ia`(*n*+1) is taken to be equal to the number of non-zeros in A, plus one. Refer to `rowIndex` array description in Sparse Matrix Storage Format for more details.
The array `ia` is also accessed in all phases of the solution process. Note that the row and columns numbers start from 1.

`ja` `INTEGER`
Array. `ja(*)`contains column indices of the sparse matrix *A* stored in compressed sparse row format. The indices in each row must be sorted in increasing order.
The array `ja` is also accessed in all phases of the solution process. For symmetric and structurally symmetric matrices it is assumed that zero diagonal elements are also stored in the list of nonzeros in *a* and `ja`. For symmetric matrices, the solver needs only the upper triangular part of the system as is shown for `columns` array in Sparse Matrix Storage Format.

perm `INTEGER`
Array, dimension (*n*). Holds the permutation vector of size *n*.
The array *perm* is defined as follows. Let A be the original matrix and $B = PAP^T$ be the permuted matrix. Row (column) `i` of A is the *perm*(`i`) row (column) of B. The numbering of the array must start by 1 and it must describe a permutation.

On entry, you can apply your own fill-in reducing ordering to the solver. The permutation vector *perm* is only accessed if *iparm*(5) = 1.

nrhs `INTEGER`.
Number of right-hand sides that need to be solved for.

iparm `INTEGER`
Array, dimension (64). This array is used to pass various parameters to PARDISO and to return some useful information after the execution of the solver.
If *iparm*(1) = 0, then PARDISO fills *iparm*(2), and *iparm*(4) through *iparm*(64) with default values and uses them. Note that there is no default values for *iparm*(3) and this value must always be supplied by the user, whether *iparm*(1) is 0 or 1.

Individual components of the *iparm* array are described below (some of them in the *Output Parameters* section).

iparm(1)

If *iparm*(1) = 0 on entry, then *iparm*(2) and *iparm*(4) through *iparm*(64) are filled with default values, otherwise the user has to supply all values in *iparm* from *iparm*(2) to *iparm*(64).

iparm(2)

iparm(2) controls the fill-in reducing ordering for the input matrix. If *iparm*(2) is 0, then the minimum degree algorithm is applied [5], if *iparm*(2) is 2, the solver uses the nested dissection algorithm from the METIS package [2].
The default value of *iparm*(2) is 2.

iparm(3)

iparm(3) must contain the number of processors that are available for the parallel execution. The number must be equal to the OpenMP environment variable

OMP_NUM_THREADS.

CAUTION. If the user has not explicitly set OMP_NUM_THREADS, then this value can be set by the operating system to the maximal numbers of processors on the system. It is therefore always recommended to control the parallel execution of the solver by explicitly setting OMP_NUM_THREADS. If less processors are available than specified, the execution may slow down instead of speeding up.

There is no default value for *iparm*(3).

iparm(4)

This parameter controls preconditioned CGS [11] for unsymmetric or structural symmetric matrices and Conjugate-Gradients for symmetric matrices.
iparm(4) has the form

iparm(4)= $10*L + K$

The values K and L have the following meaning

Value K:

Value of ***K***	Description
0	The factorization is always computed as required by *phase*
1	CGS iteration replaces the computation of *LU*. The preconditioner is *LU* that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.
2	CG iteration for symmetric matrices replaces the computation of *LU*. The preconditioner is *LU* that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.

Value L:

The value L controls the stopping criterion of the Krylow-Subspace iteration:

$\varepsilon_{CGS} = 10^{-L}$ is used in the stopping criterion

$$||dx_i|| / ||dx_1|| < \varepsilon_{CGS}$$

with $||dx_i|| = ||(LU)^{-1} r_i||$ and r_i is the residuum at iteration i of the preconditioned Krylow-Subspace iteration.

Strategy: A maximum number of 150 iterations is fixed by expecting that the iteration will converge before consuming half the factorization time. Intermediate convergence rates and residuum excursions are checked and can terminate the iteration process. If `phase` =23, then the factorization for a given *A* is automatically recomputed in these cases where the Krylow-Subspace iteration failed, and the corresponding direct solution is returned. Otherwise the solution from the preconditioned Krylow-Subspace iteration is returned. Using `phase` =33 results in an error message (`error` =4) if the stopping criteria for the Krylow-Subspace iteration can not be reached. More information on the failure can be obtained from `iparm`(20).

The default is `iparm`(4)=0, and other values are only recommended for an advanced user. `iparm`(4) must be greater or equal to zero.

Examples:

***iparm*(4)**	**Description**
31	*LU*-preconditioned CGS iteration with a stopping criterion of 10^{-3} for unsymmetric matrices
61	*LU*-preconditioned CGS iteration with a stopping criterion of 10^{-6} for unsymmetric matrices
62	*LU*-preconditioned CGS iteration with a stopping criterion of 10^{-6} for symmetric matrices

iparm(5)

In `iparm`(5), the user can apply his own fill reducing permutation instead of the integrated multiple-minimum degree or nested dissection algorithms.

This option may be useful for testing reordering algorithms or adapting the code to special applications problems (for instance, to move zero diagonal elements to the end PAP^T). For definition of the permutation, see description of the `perm` parameter.

The default value of `iparm`(5) is 0.

iparm(6)

On entry, if `iparm`(6) is 0 (which is the default), then the array `x` contains the solution and the value of `b` is not changed. If `iparm`(6) is 1, then the solver will store the solution on the right hand side `b`.

Note that the array `x` is always used. The default value of `iparm`(6) is 0.

iparm(7)

This value is not referenced. Reserved for future use.

iparm(8)

On input to the iterative refinement step, *iparm*(8) should be set to the maximum number of iterative refinement steps that the solver will perform. Iterative refinement will stop if a satisfactory level of accuracy of the solution in terms of backward error has been achieved. The solver will not perform more than the absolute value of *iparm*(8) steps of iterative refinement and will stop the process if a satisfactory level of accuracy of the solution in terms of backward error has been achieved.
The default value for *iparm*(8) is 0.

Note that if *iparm*(8) < 0, the accumulation of the residuum is using enhanced precision real and complex data types. Perturbed pivots result in iterative refinement (independent of *iparm*(8)=0) and the iteration number executed is reported on *iparm*(20).

iparm(9)

This value is reserved for future use. Value must be set to 0.

iparm(10)

On entry, *iparm*(10) instructs PARDISO how to handle small pivots or zero pivots for unsymmetric matrices (*mtype* =11 or *mtype* =13). For these matrices the solver uses a complete supernode pivoting approach. When the factorization algorithm reaches a point where it cannot factorize the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [4]. The magnitude of the potential pivot is tested against a constant threshold of

$$\varepsilon = \alpha \cdot \|A_2\|_\infty ,$$

where $\varepsilon = 10^{-iparm(10)}$ and $\|PP_{MPS}D_rAD_cP\|_\infty$ is the infinity norm of the scaled and permuted matrix A. Any tiny pivots encountered during elimination are set to the $\mathrm{sign}(l_{ii}) \cdot \varepsilon \cdot \|A_2\|_\infty$ - this trades off some numerical stability for the ability to keep pivots from getting too small. Small pivots are therefore perturbed with
$\varepsilon = 10^{-iparm(10)}$.
The default value of *iparm*(10) is 13 and therefore $\varepsilon = 10^{-13}$.

iparm(11)

PARDISO uses a maximum weight matching algorithm to permute large elements on the diagonal and to scale the matrix so that the diagonal elements are equal to 1 and the absolute value of the off-diagonal entries are less or equal to 1. This method is only applied to unsymmetric matrices (*mtype* =11 or *mtype* =13) and, by default, indicated with *iparm*(11)=1, this option is always turned on. Otherwise, the scalings are omitted.
The default value of *iparm*(11) is 1.

iparm(12)

This value is reserved for future use. Value must be set to 0.

iparm(13)

This value is reserved for future use. Value must be set to 0.

iparm(18)

The solver will report the numbers of nonzeros on the factors if *iparm*(18) < 0 on entry.

The default value of *iparm*(18) is 0.

iparm(19)

If *iparm*(19)< 0 on entry, the solver will report MFlop (10^6) that are necessary to factor the matrix *A*. This will increase the reordering time.

The default value of *iparm*(19) is 0.

msglvl INTEGER.
Message level information. If *msglvl* = 0 then PARDISO generates no output, if *msglvl* = 1 the solver prints statistical information in the file pardiso.stat.#*nproc*.

b REAL/COMPLEX
Array, dimension (*n*, *nrhs*). On entry, contains the right hand side vector/matrix *B*. Note that *b* is only accessed in the solution phase.

Output Parameters

pt On exit, contains internal address pointers.

iparm On output, some *iparm* values will contain useful information, for example, numbers of nonzeros in the factors, and so on.

iparm(14)

After factorization, *iparm*(14) contains the number of perturbed pivots during the elimination process for *mtype* =11 or *mtype* =13.

iparm(15)

On output, *iparm*(15) provides the user with the total peak memory in KBytes that the solver needed during the analysis and symbolic factorization phase. This value is only computed in phase 1.

iparm(16)

On output, *iparm*(16) provides the user with the permanent memory in KBytes that the solver needed from the analysis and symbolic factorization phase in the factorization and solve phases. This value is only computed in phase 1.

iparm(17)

On output, *iparm*(17) provides the user with the total double precision memory consumption (KBytes) of the solver for the factorization and solve phases. This value is only computed in phase 2.

Note that the total peak memory solver consumption is max(*iparm*(15), *iparm*(16)+*iparm*(17)).

iparm(18)

On output, the numbers of nonzeros on the factors are returned to the user.

iparm(19)

Number of operations in MFlop (10^6 operations) that are necessary to factor the matrix *A* are returned to the user.

iparm(20)

The value is used to give CG/CGS diagnostics (for example, the number of iterations and cause of failure):

If *iparm*(20) > 0, CGS succeeded, and the number of iterations executed are reported in *iparm*(20).

If *iparm*(20) < 0, iterations executed, but CG/CGS failed. The error report details in *iparm*(20) are of the form:

iparm(20) = - it_cgs*10 - cgs_error.

If *phase* was 23, then the factors *L*, *U* have been recomputed for the matrix *A* and the error flag *error* should be zero in case of a successful factorization. If *phase* was 33, then *error* = -4 will signal the failure.

Description of cgs_error is given in the below table:

cgs_error	Description
1	too large fluctuations of the residuum
2	$\|\|dx_{max\_it\_cgs/2}\|\|$ too large (slow convergence)
3	stopping criterion not reached at max_it_cgs
4	perturbed pivots caused iterative refinement
5	factorization is too fast for this matrix. It is better to use the factorization method with *iparm*(4) = 0

iparm(21) to ***iparm(64)***

These values are reserved for future use. Value must be set to 0.

b On output, the array is replaced with the solution if *iparm*(6) = 1.

x REAL/COMPLEX
Array, dimension (*n*, *nrhs*). On output, contains solution if *iparm*(6)= 0.
Note that *x* is only accessed in the solution phase.

error INTEGER.
The error indicator according to the below table:

error	Information
0	no error
-1	input inconsistent
-2	not enough memory
-3	reordering problem
-4	zero pivot, numerical factorization problem
-5	unclassified (internal) error
-6	preordering failed (matrix types 11, 13 only)
-7	diagonal matrix problem

Direct Sparse Solver (DSS) Interface Routines

The Intel MKL supports an alternative to PARDISO interface for the direct sparse solver referred to here as DSS interface. The DSS interface implements a group of user-callable routines that are used in the step-by-step solving process and exploits the general scheme described in Linear Solvers Basics for solving sparse systems of linear equations. This interface also includes one routine for gathering statistics related to the solving process and an auxiliary routine for passing character strings from Fortran routines to C routines.

The solving process is conceptually divided into six phases, as shown in Table 8-1 which lists the names of the routines, grouped by phase, and describes their general use.

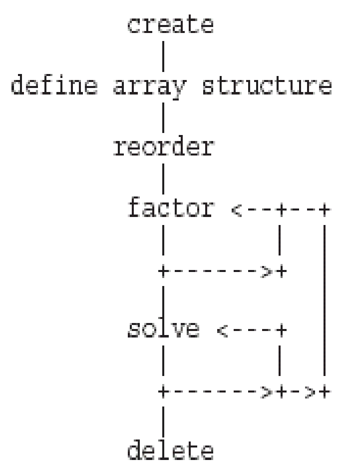
Table 8-1 DSS Interface Routines

Routine	Description
`dss_create`	Initializes the solver and creates the basic data structures necessary for the solver. This routine must be called before any other DSS routine.
`dss_define_structure`	Used to inform the solver of the locations of the non-zero elements of the array.
`dss_reorder`	Based on the non-zero structure of the matrix, this routine computes a permutation vector to reduce fill-in during the factoring process.
`dss_factor_real` `dss_factor_complex`	Computes the *LU*, LDT^t or LL^T factorization of a real or complex matrix.
`dss_solve_real` `dss_solve_complex`	Computes the solution vector for a system of equations based on the factorization computed by the previous phase.
`dss_delete`	Deletes all of the data structures created during the solutions process.
`dss_statistics`	Returns statistics about various phases of the solving process. Used to gather statistics in the following areas: time taken to do reordering, time taken to do factorization, problem solving duration, determinant of a matrix, inertia of a matrix, and number of floating point operations taken during factorization. Can be invoked at any phase of the solving process after the "reorder" phase, but before the "delete" phase. Note that appropriate argument(s) must be supplied to this routine to correspond to phase at which it is invoked.
`mkl_cvt_to_null_terminated_str`	Used to pass character strings from Fortran routines to C routines

To find a single solution vector for a single system of equations with a single right hand side, the Intel MKL DSS interface routines are invoked in the order in which they are listed in Table 8-1, with the exception of `dss_statistics`, which is invoked as described in the table.

However, in certain applications it is necessary to produce solution vectors for multiple right-hand sides for a given factorization and/or factor several matrices with the same non-zero structure. Consequently, it is necessary to be able to invoke the Intel MKL sparse routines in an order other than listed in the table. The following diagram in Figure 8-2 indicates the typical order(s) in which the DSS interface routines can be invoked.

Figure 8-2 Typical order for invoking DSS interface routines



You can find example code that uses DSS interface routines to solve systems of linear equations in Direct Sparse Solver Examples section in the appendix.

Interface Description

As noted in Memory Allocation and Handles section, each DSS routine either reads or writes an opaque data object called a handle. Because the declaration of a handle varies from language to language, it is declared as being of type `MKL_DSS_HANDLE` in this documentation. You can refer to Memory Allocation and Handles to determine the correct method for declaring a handle argument.

All other types in this documentation refer to the standard Fortran types, `INTEGER`, `REAL`, `COMPLEX`, `DOUBLE PRECISION`, and `DOUBLE COMPLEX`.

C and C++ programmers should refer to Calling Direct Sparse Solver Routines From C/C++ for information on mapping Fortran types to C/C++ types.

Routine Options

All of the DSS routines have an integer argument (below referred to as *`opt`*) for passing various options to the routines. The permissible values for *`opt`* should be specified using only the symbol constants defined in the language-specific header files (see Implementation Details). All of the routines accept options for setting the message and termination level as described in Table 8-2. Additionally, all routines accept the option `MKL_DSS_DEFAULTS`, which establishes the documented default options for each DSS routine.

Table 8-2 Symbolic Names for the Message and Termination Level Options

Message Level	Termination Level
MKL_DSS_MSG_LVL_SUCCESS	MKL_DSS_TERM_LVL_SUCCESS
MKL_DSS_MSG_LVL_INFO	MKL_DSS_TERM_LVL_INFO
MKL_DSS_MSG_LVL_WARNING	MKL_DSS_TERM_LVL_WARNING
MKL_DSS_MSG_LVL_ERROR	MKL_DSS_TERM_LVL_ERROR
MKL_DSS_MSG_LVL_FATAL	MKL_DSS_TERM_LVL_FATAL

The settings for message and termination level can be set on any call to a DSS routine. However, once set to a particular level, they remain at that level until they are changed in another call to a DSS routine.

Users can specify multiple options to a DSS routine by adding the options together. For example, to set the message level to debug and the termination level to error for all DSS routines, use the call:

```
CALL dss_create( handle, MKL_DSS_MSG_LVL_INFO + MKL_DSS_TERM_LVL_ERROR)
```

User Data Arrays

Many of the DSS routines take arrays of user data as input. For example, user arrays are passed to the routine `dss_define_structure` to describe the location of the non-zero entries in the matrix. In order to minimize storage requirements and improve overall run-time efficiency, the Intel MKL DSS routines do not make copies of the user input arrays.

WARNING. Users cannot modify the contents of these arrays after they are passed to one of the solver routines.

dss_create

Initializes the solver.

Syntax

`dss_create (`*`handle`*`, `*`opt`*`)`

Input Arguments

`opt` — `INTEGER`. Options passing argument. The default value is `MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR` .

Output Arguments

`handle` — Data object of `MKL_DSS_HANDLE` type (see Interface Description).

Description

The routine `dss_create` is called to initialize the solver. After the call to `dss_create`, all subsequent invocations of Intel MKL DSS routines should use the value of handle returned by `dss_create`.

WARNING. Do not write the value of handle directly.

Return Values

MKL_DSS_SUCCESS

MKL_DSS_INVALID_OPTION

MKL_DSS_OUT_OF_MEMORY

dss_define_structure

Communicates to the solver locations of non-zero elements in the matrix.

Syntax

```
dss_define_structure (handle, opt, rowIndex, nRows, nCols, columns,
    nNonZeros);
```

Input Arguments

opt	INTEGER. Option passing argument. The default option for the matrix structure is MKL_DSS_SYMMETRIC.
rowIndex	INTEGER. Array of size min(*nRows*, *nCols*)+1. Defines the location of non-zero entries in the matrix.
nRows	INTEGER. Number of rows in the matrix.
nCols	INTEGER. Number of columns in the matrix.
columns	INTEGER. Array of size *nNonZeros*. Defines the location of non-zero entries in the matrix.
nNonZeros	INTEGER. Number of non-zero elements in the matrix.

Output Arguments

handle	Data object of MKL_DSS_HANDLE type (see Interface Description).

Description

The routine dss_define_structure communicates to the solver the locations of the *nNonZeros* number of non-zero elements in a matrix of size *nRows* by *nCols*.
Note that currently Intel MKL DSS software only operates on square matrices, so *nRows* must be equal to *nCols*.

To communicate the locations of non-zeros in the matrix, do the following:

1. Define the general non-zero structure of the matrix by specifying one of the following values for the options argument *opt*:

   MKL_DSS_SYMMETRIC_STRUCTURE

   MKL_DSS_SYMMETRIC

   MKL_DSS_NON_SYMMETRIC

2. Provide the actual locations of the non-zeros by means of the arrays *rowIndex* and *columns* (see Sparse Matrix Storage Format).

NOTE. Currently, DSS software in Intel MKL does not directly support non-symmetric matrices. Instead, when the MKL_DSS_NON_SYMMETRIC option is specified, the solver will convert non-symmetric matrices into symmetrically structured matrices by adding zeros in the appropriate place.

Return Values

MKL_DSS_SUCCESS

MKL_DSS_STATE_ERR

MKL_DSS_INVALID_OPTION

MKL_DSS_COL_ERR

MKL_DSS_NOT_SQUARE

MKL_DSS_TOO_FEW_VALUES

MKL_DSS_TOO_MANY_VALUES

dss_reorder

Computes permutation vector that minimizes the fill-in during the factorization phase.

Syntax

```
dss_reorder (handle, opt, perm)
```

Input Arguments

opt INTEGER. Option passing argument. The default option for the permutation type is MKL_DSS_AUTO_ORDER.

perm INTEGER. Array of length *nRows*. Contains a user-defined permutation vector (accessed only if *opt* contains MKL_DSS_MY_ORDER).

Output Arguments

handle Data object of MKL_DSS_HANDLE type (see Interface Description).

Description

If *opt* contains the options MKL_DSS_AUTO_ORDER, then dss_reorder computes a permutation vector that minimizes the fill-in during the factorization phase. For this option, the *perm* array is never accessed.

If *opt* contains the option MKL_DSS_MY_ORDER, then the array *perm* is considered to be a permutation vector supplied by the user. In this case, the array *perm* is of length *nRows*, where *nRows* is the number of rows in the matrix as defined by the previous call to dss_define_structure.

Return Values

MKL_DSS_SUCCESS

MKL_DSS_STATE_ERR

MKL_DSS_INVALID_OPTION

MKL_DSS_OUT_OF_MEMORY

dss_factor_real
dss_factor_complex

Compute the factorization of the matrix with previously specified location.

Syntax

```
dss_factor_real (handle, opt, rValues)
dss_factor_complex (handle, opt, cValues)
```

Input Arguments

handle	Data object of `MKL_DSS_HANDLE` type (see Interface Description).
opt	`INTEGER`. Option passing argument. The default option for the matrix type is `MKL_DSS_POSITIVE_DEFINITE`.
rValues	`DOUBLE PRECISION`. Array of size *nNonZeros*. Contains real non-zero elements of the matrix.
cValues	`DOUBLE COMPLEX`. Array of size *nNonZeros*. Contains complex non-zero elements of the matrix.

Description

These routines compute the factorization of the matrix whose non-zero locations were previously specified by a call to `dss_define_structure` and whose non-zero values are given in the array *rValues* or *cValues*. The arrays *rValues* and *cValues* are assumed to be of length *nNonZeros* as defined in a previous call to `dss_define_structure`.

The *opt* argument should contain one of the following options:

`MKL_DSS_POSITIVE_DEFINITE`,
`MKL_DSS_INDEFINITE`,
`MKL_DSS_HERMITIAN_POSITIVE_DEFINITE`,
`MKL_DSS_HERMITIAN_INDEFINITE` ,

depending on whether the non-zero values in *rValues* and *cValues* describe a positive definite, indefinite, or Hermitian matrix.

Return Values

`MKL_DSS_SUCCESS`

`MKL_DSS_STATE_ERR`

`MKL_DSS_INVALID_OPTION`

`MKL_DSS_OPTION_CONFLICT`

`MKL_DSS_OUT_OF_MEMORY`

`MKL_DSS_ZERO_PIVOT`

dss_solve_real
dss_solve_complex

Compute the corresponding solutions vector and place it in the output array.

Syntax

```
dss_solve_real (handle, opt, rRhsValues, nRhs, rSolValues)
dss_solve_complex (handle, opt, cRhsValues, nRhs, cSolValues)
```

Input Arguments

handle	Data object of `MKL_DSS_HANDLE` type (see Interface Description).
opt	`INTEGER`. Option passing argument.
nRhs	`INTEGER`. Number of the right-hand sides in the linear equation.
rRhsValues	`DOUBLE PRECISION.` Array of size *nRows* by *nRhs*. Contains real right-hand side vectors.
cRhsValues	`DOUBLE COMPLEX.` Array of size *nRows* by *nRhs*. Contains complex right-hand side vectors.

Output Arguments

rSolValues	`DOUBLE PRECISION.` Array of size *nCols* by *nRhs*. Contains real solution vectors.
cSolValues	`DOUBLE COMPLEX`. Array of size *nCols* by *nRhs*. Contains complex solution vectors.

Description

For each right hand side column vector defined in *?RhsValues* (where *?* is one of *r* or *c*), these routines compute the corresponding solutions vector and place it in the array *?SolValues*.

The lengths of the right-hand side and solution vectors, *nCols* and *nRows* respectively, are assumed to have been defined in a previous call to `dss_define_structure`.

Return Values

`MKL_DSS_SUCCESS`

MKL_DSS_STATE_ERR

MKL_DSS_INVALID_OPTION

MKL_DSS_OUT_OF_MEMORY

dss_delete

Deletes all of data structures created during the solutions process.

Syntax

`dss_delete (handle, opt)`

Input Arguments

opt	INTEGER. Options passing argument. The default value is MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR.

Output Arguments

handle	Data object of MKL_DSS_HANDLE type (see Interface Description).

Description

The routine `dss_delete` is called to delete all of the data structures created during the solutions process.

Return Values

MKL_DSS_SUCCESS

MKL_DSS_INVALID_OPTION

MKL_DSS_OUT_OF_MEMORY

dss_statistics

Returns statistics about various phases of the solving process.

Syntax

```
dss_statistics (handle, opt, statArr, retValues)
```

Input Arguments

handle	Data object of `MKL_DSS_HANDLE` type (see Interface Description).
opt	`INTEGER`. Options passing argument.
statArr	`STRING`. Input string that defines the type of the returned statistics. Can include one or more of the following string constants (case of the input string has no effect):

`ReorderTime`	Amount of time taken to do the reordering.
`FactorTime`	Amount of time taken to do the factorization.
`SolveTime`	Amount of time taken to solve the problem after factorization.
`Determinant`	Determinant of the matrix *A*. For real matrices, determinant is returned as `det_pow`, `det_base` in two consecutive return array locations, where: $1.0 \le abs(det\_base) < 10.0$ and $determinant = det\_base \cdot 10^{det\_pow}$. For complex matrices, determinant is returned as *det_pow*, *det_re*, *det_im* in three consecutive return array locations, where: $1.0 \le abs(det\_re) + abs(det\_im) < 10.0$ and $determinant = (det\_re, det\_im) \cdot 10^{det\_pow}$.
`Inertia`	Inertia of a real symmetric matrix is defined to be a triplet of nonnegative integers (p,n,z) where p is a number of positive eigenvalues, n is number of negative eigenvalues, and z is number of zero eigenvalues.

`Inertia` will be returned as three consecutive return array locations as *p*,*n*,*z*.

Computing `Inertia` is only recommended for stable matrices. Unstable matrices can lead to incorrect results.

`Inertia` of a *kxk* real symmetric positive definite matrix is always (*k*,0,0). Therefore `Inertia` is returned only in cases of real symmetric indefinite matrices. For all other matrix types, an error message is returned.

`Flops` Number of floating point operations performed during factorization.

NOTE. To avoid problems in passing strings from Fortran to C, Fortran users must call the `mkl_cvt_to_null_terminated_str` routine before calling `dss_statistics`. Refer to the description of `mkl_cvt_to_null_terminated_str` for details.

Output Arguments

`retValues` `DOUBLE PRECISION`. Value of the statistics returned.

Description

The `dss_statistics` routine returns statistics about various phases of the solving process. Use this routine to gather statistics in the following areas:

— time taken to do reordering,

— time taken to do factorization,

— problem solving duration,

— determinant of a matrix,

— inertia of a matrix,

— number of floating point operations taken during factorization.

Statistics are returned corresponding to the specified input string. The value of the statistics is returned in double precision in a return array allocated by user.

For multiple statistics, string constants separated by commas can be used as input. Return values are put into the return array in the same order as specified in the input string.

Statistics should only be requested at appropriate stages of the solving process. For example, inquiring about `FactorTime` before a matrix is factored will lead to errors.

The following table shows the point at which each statistic can be called:

Table 8-3 Statistics Calling Sequences

Type of Statistics	When to Call
`ReorderTime`	After `dss_reorder` is completed successfully.
`FactorTime`	After `dss_factor_real` or `dss_factor_complex` is completed successfully.
`SolveTime`	After `dss_solve_real` or `dss_solve_complex` is completed successfully.
`Determinant`	After `dss_factor_real` or `dss_factor_complex` is completed successfully.
`Inertia`	After `dss_factor_real` is completed successfully and matrix is real, symmetric, and indefinite.
`Flops`	After `dss_factor_real` or `dss_factor_complex` is completed successfully.

The example below illustrates the use of the `dss_statistics` routine.

Example 8-1 Finding "time used to reorder" and "inertia" of a matrix.

To find these values, call
`dss_statistics(`*`handle`*`, `*`opt`*`, `*`statArr`*`, `*`retValues`*`)`,
where *`statArr`* is `"ReorderTime,Inertia"`

In this example, *`retValues`* will have the following values:

`retValue[0]` Time to reorder.

`retValue[1]` Positive Eigenvalues.

`retValue[2]` Negative Eigenvalues.

`retValue[3]` Zero Eigenvalues.

Return Values

`MKL_DSS_SUCCESS`

`MKL_DSS_STATISTICS_INVALID_MATRIX`

`MKL_DSS_STATISTICS_INVALID_STATE`

`MKL_DSS_STATISTICS_INVALID_STRING`

mkl_cvt_to_null_terminated_str

Passes character strings from Fortran routines to C routines.

Syntax

`mkl_cvt_to_null_terminated_str (`*`destStr`*`, `*`destLen`*`, `*`srcStr`*`)`

Input Arguments

`destLen` `INTEGER`. Length of the output array *`destStr`*.

`srcStr` `STRING`. Input string.

Output Arguments

`destStr` `INTEGER`. One-dimensional array of integer.

Description

The routine `mkl_cvt_to_null_terminated_str` is used to pass character strings from Fortran routines to C routines. The strings are converted into integer arrays before being passed to C. Using this routine avoids the problems that can occur on some platforms when passing strings from Fortran to C. The use of this routine is highly recommended.

Implementation Details

Several aspects of the Intel MKL DSS interface are platform-specific and language-specific. In order to promote portability across platforms and ease of use across different languages, users are encouraged to include one of the Intel MKL DSS language-specific header files. Currently, there are three language specific header files:

- `mkl_dss.f77` for F77 programs
- `mkl_dss.f90` for F90 programs
- `mkl_dss.h` for C programs

These language-specific header files define symbolic constants for error returns, function options, certain defined data types, and function prototypes.

NOTE. It is strongly recommended that you refer to the constants for options, error returns, and message severities **only** by the symbolic names that are defined in the header files. Use of the Intel MKL DSS software without including one of the above header files is not supported.

Memory Allocation and Handles

In order to make the Intel MKL DSS routines as easy to use as possible, the routines do not require the user to allocate any temporary working storage. Any storage required by the solver (that is not a user input) is allocated by the solver itself. In order to allow multiple users to access the solver simultaneously, the solver keeps track of the storage allocated for a particular application by using an opaque data object called a **handle**.

Each of the Intel MKL DSS routines either creates, uses or deletes a handle. Consequently, user programs must be able to allocate storage for a handle. The exact syntax for allocating storage for a handle varies from language to language. To help standardize the handle declarations, the language-specific header files declare constants and defined data types that should be used when declaring a handle object in user code.

Fortran 90 programmers should declare a handle as:

```
INCLUDE "mkl_dss.f90"
TYPE(MKL_DSS_HANDLE) handle
```

C and C++ programmers should declare a handle as:

```
#include "mkl_dss.h"
_MKL_DSS_HANDLE_t handle;
```

Fortran 77 programmers using compilers that support eight byte integers, should declare a handle as:

```
INCLUDE "mkl_dss.f77"
INTEGER*8 handle
```

Otherwise they should replace `INTEGER*8` with `DOUBLE PRECISION`.

In addition to the necessary definition for the correct declaration of a handle, the include file also defines the following:

- function prototypes for languages that support prototypes
- symbolic constants that are used for the error returns
- user options for the solver routines
- message severity

Calling Direct Sparse Solver Routines From C/C++

The calling interface for all of the Intel MKL DSS routines is designed to be used easily from Fortran 77 or Fortran 90. However, any of the DSS routines can be invoked directly from C or C++ if users are familiar with the inter-language calling conventions of their platforms. These conventions include, but are not limited to, the argument passing mechanisms for the language, the data type mappings from Fortran to C/C++ and how Fortran external names are decorated on the platform.

In order to promote portability and to avoid having most users deal with these issues, the C header file `mkl_dss.h` provides a set of macros and type definitions that are intended to hide the inter-language calling conventions and provide an interface to the DSS that appears natural for C/C++.

For example, consider a hypothetical library routine, `foo`, that takes real vector of length *n*, and returns an integer status. Fortran users would access such a function as:

```
INTEGER n, status, foo
REAL x(*)
status = foo(x, n)
```

As noted above, for C users to invoke `foo`, they would need to know what C data types correspond to Fortran types `INTEGER` and `REAL`; what argument passing mechanism the Fortran compiler uses; and what, if any, name decoration the is performed by the Fortran compiler when generating the external symbol `foo`.

However, by using the C specific header file `mkl_dss.h`, the invocation of `foo`, within a C program would look like:

```
#include "mkl_dss.h"
_INTEGER_t i, status;
_REAL_t x[];
status = foo( x, i );
```

Note that in the above example, the header file `mkl_dss.h` provides definitions for the types `_INTEGER_t` and `_REAL_t` that correspond to the Fortran types `INTEGER` and `REAL`.

In order to ease the use of DSS routines from C and C++, the general approach of providing C definitions of Fortran types is used throughout the library. Specifically, if an argument or result from a direct sparse solver is documented as having the Fortran language specific type `XXX`, then the C and C++ header files provide an appropriate C language type definitions for the name `_XXX_t`.

Caveat for C Users

One of the key differences between C/C++ and Fortran is the argument passing mechanisms for the languages: Fortran programs use pass-by-reference semantics and C/C++ programs use pass-by-value semantics. In the example in the previous section, the header file, `mkl_dss.h`, attempts to hide this difference, by defining a macro, `foo` that takes the address of the appropriate arguments. For example, on Tru64 UNIX, `mkl_dss.h` would define the macro as:

```
#define foo(a,b) foo_((a), &(b))
```

An important point to note when using the macro form of `foo` is how it deals with constants. If we write `foo( x, 10 )`, this is translated into `foo_( x, &10 )`. In a strictly ANSI compliant C compiler, it is not permissible to take the address of a constant, so a strictly conforming program would look like:

```
_INTEGER_t iTen = 10;
_REAL_t * x;
status = foo( x, iTen );
```

However, some C compilers in a non-ANSI standard mode allow taking the address of a constant for ease of use with Fortran programs. Thus, the form shown as `foo( x, 10 )` is acceptable for these compilers.

Vector Mathematical Functions

9

This chapter describes Vector Mathematical Functions Library (VML), which is designed to compute elementary functions on vector arguments. VML is an integral part of the Intel® MKL Kernel Library and the VML terminology is used here for simplicity in discussing this group of functions.

VML includes a set of highly optimized implementations of certain computationally expensive core mathematical functions (power, trigonometric, exponential, hyperbolic etc.) that operate on vectors.

Application programs that might significantly improve performance with VML include nonlinear programming software, integrals computation, and many others.

VML functions are divided into the following groups according to the operations they perform:

- "VML Mathematical Functions" compute values of elementary functions (such as sine, cosine, exponential, logarithm and so on) on vectors with unit increment indexing.
- "VML Pack/Unpack Functions" convert to and from vectors with positive increment indexing, vector indexing and mask indexing (see Appendix B for details on vector indexing methods).
- "VML Service Functions" allow the user to set /get the accuracy mode, and set/get the error code.

VML mathematical functions take an input vector as argument, compute values of the respective elementary function element-wise, and return the results in an output vector.

Data Types and Accuracy Modes

Mathematical and pack/unpack vector functions in VML have been implemented for vector arguments of single and double precision real data. Both Fortran- and C-interfaces to all functions, including VML service functions, are provided in the library. The differences in naming and calling the functions for Fortran- and C-interfaces are detailed in the "Function Naming Conventions" section below.

Each vector function from VML (for each data format) can work in two modes: High Accuracy (HA) and Low Accuracy (LA). For many functions, using the LA version will improve performance at the cost of accuracy.
For some cases, the advantage of relaxing the accuracy improves performance very little so the same function is employed for both versions. Error behavior depends not only on whether the HA or LA version is chosen, but also depends on the processor on which the software runs.
In addition, special value behavior may differ between the HA and LA versions of the functions. Any information on accuracy behavior can be found in the Intel *MKL Release Notes*.

Switching between the two modes (HA and LA) is accomplished by using `vmlSetMode(mode)` (see Table 9-11). The function `vmlGetMode()` will return the currently used mode. The High Accuracy mode is used by default.

Function Naming Conventions

Full names of all VML functions include only lowercase letters for Fortran-interface, whereas for C-interface names the lowercase letters are mixed with uppercase.

VML mathematical and pack/unpack function full names have the following structure:

`v <p> <name> <mod>`

The initial letter `v` is a prefix indicating that a function belongs to VML.
The `<p>` field is a precision prefix that indicates the data type:

`s`	`REAL` for Fortran–interface, or `float` for C–interface
`d`	`DOUBLE PRECISION` for Fortran–interface, or `double` for C–interface.

The `<name>` field indicates the function short name, with some of its letters in uppercase for C-interface (see Tables 7-2, 7-8).

The `<mod>` field (written in uppercase for C-interface) is present in pack/unpack functions only; it indicates the indexing method used:

`i`	indexing with positive increment
`v`	indexing with index vector
`m`	indexing with mask vector.

VML service function full names have the following structure:

`vml <name>`

where `vml` is a prefix indicating that a function belongs to VML, and `<name>` is the function short name, which includes some uppercase letters for C-interface (see Table 9-10).
To call VML functions from an application program, use conventional function calls. For example, the VML exponential function for single precision data can be called as

`call vsexp ( n, a, y )` for Fortran–interface, or

`vsExp ( n, a, y );` for C–interface.

Functions Interface

The interface to VML functions includes function full names and the arguments list. The Fortran- and C-interface descriptions for different groups of VML functions are given below. Note that some functions (`Div`, `Pow`, and `Atan2`) have two input vectors *a* and *b* as their arguments, while `SinCos` function has two output vectors *y* and *z*.

VML Mathematical Functions:

Fortran:

```
call v<p><name>( n, a, y )
call v<p><name>( n, a, b, y )
call v<p><name>( n, a, y, z )
```

C:

```
v<p><name>( n, a, y );
v<p><name>( n, a, b, y );
v<p><name>( n, a, y, z );
```

Pack Functions:

Fortran:

```
call v<p>packi( n, a, inca, y )
call v<p>packv( n, a, ia, y )
call v<p>packm( n, a, ma, y )
```

C:

```
v<p>PackI( n, a, inca, y );
v<p>PackV( n, a, ia, y );
v<p>PackM( n, a, ma, y );
```

Unpack Functions:

Fortran:

```
call v<p>unpacki( n, a, y, incy )
call v<p>unpackv( n, a, y, iy )
call v<p>unpackm( n, a, y, my )
```

C:

```
v<p>UnpackI( n, a, y, incy );
v<p>UnpackV( n, a, y, iy );
v<p>UnpackM( n, a, y, my );
```

Service Functions:

Fortran:

```
oldmode  = vmlsetmode( mode )
mode     = vmlgetmode( )
olderr   =  vmlseterrstatus ( err )
err      =  vmlgeterrstatus( )
olderr   =  vmlclearerrstatus( )
oldcallback = vmlseterrorcallback( callback )
callback    = vmlgeterrorcallback( )
oldcallback =  vmlclearerrorcallback( )
```

C:

```
oldmode = vmlSetMode( mode );
mode    = vmlGetMode( void);
olderr  = vmlSetErrStatus ( err );
err     = vmlGetErrStatus(void);
olderr  = vmlClearErrStatus(void);
oldcallback = vmlSetErrorCallBack(callback );
callback    = vmlGetErrorCallBack( void );
oldcallback = vmlClearErrorCallBack(void );
```

Input Parameters:

n	number of elements to be calculated
a	first input vector
b	second input vector
inca	vector increment for the input vector *a*
ia	index vector for the input vector *a*
ma	mask vector for the input vector *a*
incy	vector increment for the output vector *y*
iy	index vector for the output vector *y*
my	mask vector for the output vector *y*
err	error code
mode	VML mode
callback	address of the callback function

Output Parameters:

y	first output vector
z	second output vector
err	error code
mode	VML mode
olderr	former error code

oldmode former VML mode

oldcallback address of the former callback function

The data types of the parameters used in each function are specified in the respective function description section. All VML mathematical functions can perform in-place operations, which means that the same vector can be used as both input and output parameter. This holds true for functions with two input vectors as well, in which case one of them may be overwritten with the output vector. For functions with two output vectors, one of them may coincide with the input vector.

Vector Indexing Methods

Current VML mathematical functions work only with unit increment. Arrays with other increments, or more complicated indexing, can be accommodated by gathering the elements into a contiguous vector and then scattering them after the computation is complete.
Three following indexing methods are used to gather/scatter the vector elements in VML:

- positive increment
- index vector
- mask vector.

The indexing method used in a particular function is indicated by the indexing modifier (see the description of the `<mod>` field in "Function Naming Conventions"). For more information on indexing methods see Vector Arguments in VML in Appendix B.

Error Diagnostics

The VML library has its own error handler. The only difference for C- and Fortran- interfaces is that the Intel MKL error reporting routine `XERBLA` can be called after the Fortran- interface VML function encounters an error, and this routine gets information on `VML_STATUS_BADSIZE and VML_STATUS_BADMEM` input errors (see Table 9-13).

The VML error handler has the following properties:

1. The Error Status (`vmlErrStatus`) global variable is set after each VML function call. The possible values of this variable are shown in the Table 9-13.
2. Depending on the VML mode, the error handler function invokes:
   - `errno` variable setting. The possible values are shown in the Table 9-1.
   - writing error text information to the `stderr` stream

- raising the appropriate exception on error, if necessary
- calling the additional error handler callback function.

Table 9-1 Set Values of the errno Variable

Value of `errno`	Description
`0`	No errors are detected.
`EINVAL`	The array dimension is not positive.
`EACCES`	NULL pointer is passed.
`EDOM`	At least one of array values is out of a range of definition.
`ERANGE`	At least one of array values caused a singularity, overflow or underflow.

VML Mathematical Functions

This section describes VML functions which compute values of elementary mathematical functions on real vector arguments with unit increment.
Each function group is introduced by its short name, a brief description of its purpose, and the calling sequence for each type of data both for Fortran- and C-interfaces, as well as a description of the input/output arguments.

For all VML mathematical functions, the input range of parameters is equal to the mathematical range of definition in the set of defined values for the respective data type. Several VML functions, specifically `Div`, `Exp`, `Sinh`, `Cosh`, and `Pow`, can result in an overflow. For these functions, the respective input threshold values that mark off the precision overflow are specified in the function description section. Note that in these specifications, `FLT_MAX` denotes the maximum number representable in single precision data type, while `DBL_MAX` denotes the maximum number representable in double precision data type.

Table 9-2 lists available mathematical functions and data types associated with them.

Table 9-2 Continuous Distribution Generators

Type of Distribution	Data Types	Description
Power and Root Functions		
`Inv`	`s, d`	Inversion of the vector elements
`Div`	`s, d`	Divide elements of one vector by elements of second vector

Table 9-2 Continuous Distribution Generators (continued)

Type of Distribution	Data Types	Description
`Sqrt`	`s, d`	Square root of vector elements
`InvSqrt`	`s, d`	Inverse square root of vector elements
`Cbrt`	`s, d`	Cube root of vector elements
`InvCbrt`	`s, d`	Inverse cube root of vector elements
`Pow`	`s, d`	Each vector element raised to the specified power
`Powx`	`s, d`	Each vector element raised to the constant power
Exponential and Logarithmic Functions		
`Exp`	`s, d`	Exponential of vector elements
`Ln`	`s, d`	Natural logarithm of vector elements
`Log10`	`s, d`	Denary logarithm of vector elements
Trigonometric Functions		
`Cos`	`s, d`	Cosine of vector elements
`Sin`	`s, d`	Sine of vector elements
`SinCos`	`s, d`	Sine and cosine of vector elements
`Tan`	`s, d`	Tangent of vector elements
`Acos`	`s, d`	Inverse cosine of vector elements
`Asin`	`s, d`	Inverse sine of vector elements
`Atan`	`s, d`	Inverse tangent of vector elements
`Atan2`	`s, d`	Four-quadrant inverse tangent of elements of two vectors
Hyperbolic Functions		
`Cosh`	`s, d`	Hyperbolic cosine of vector elements
`Sinh`	`s, d`	Hyperbolic sine of vector elements
`Tanh`	`s, d`	Hyperbolic tangent of vector elements
`Acosh`	`s, d`	Inverse hyperbolic cosine (nonnegative) of vector elements
`Asinh`	`s, d`	Inverse hyperbolic sine of vector elements
Special Functions		
`Erf`	`s, d`	Error function value of vector elements
`Erfc`	`s, d`	Complementary error function value of vector elements

Inv

Performs element by element inversion of the vector.

Syntax

Fortran:

```
call vsinv( n, a, y )
call vdinv( n, a, y )
```

C:

```
vsInv( n, a, y );
vdInv( n, a, y );
```

Input Parameters

Fortran:

n	`INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.
a	`REAL, INTENT(IN)` for `vsinv` `DOUBLE PRECISION, INTENT(IN)` for `vdinv` Array, specifies the input vector `a`.

C:

n	`int`. Specifies the number of elements to be calculated.
a	`const float*` for `vsInv` `const double*` for `vdInv` Pointer to an array that contains the input vector `a`.

Output Parameters

Fortran:

y	`REAL` for `vsinv` `DOUBLE PRECISION` for `vdinv` Array, specifies the output vector `y`.

C:

y `float*` for `vsInv`
`double*` for `vdInv`
Pointer to an array that contains the output vector `y`.

Div

Performs element by element division of vector `a` *by vector* `b`*.*

Syntax

Fortran:

```
call vsdiv( n, a, b, y )
call vddiv( n, a, b, y )
```

C:

```
vsDiv( n, a, b, y );
vdDiv( n, a, b, y );
```

Input Parameters

Fortran:

n `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

a, *b* `REAL, INTENT(IN)` for `vsdiv`
`DOUBLE PRECISION, INTENT(IN)` for `vddiv`
Arrays, specify the input vectors `a` and `b`.

C:

n `int`. Specifies the number of elements to be calculated.

a, *b* `const float*` for `vsDiv`
`const double*` for `vdDiv`
Pointers to arrays that contain the input vectors `a` and `b`.

Table 9-3 Precision Overflow Thresholds for `Div` Function

Data Type	Threshold Limitations on Input Parameters
single precision	`abs(a[i]) < abs(b[i]) * FLT_MAX`
double precision	`abs(a[i]) < abs(b[i]) * DBL_MAX`

Output Parameters

Fortran:

y — `REAL` for `vsdiv`
`DOUBLE PRECISION` for `vddiv`
Array, specifies the output vector y.

C:

y — `float*` for `vsDiv`
`double*` for `vdDiv`
Pointer to an array that contains the output vector y.

Sqrt

Computes a square root of vector elements.

Syntax

Fortran:

```
call vssqrt( n, a, y )
call vdsqrt( n, a, y )
```

C:

```
vsSqrt( n, a, y );
vdSqrt( n, a, y );
```

Input Parameters

Fortran:

n `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

a `REAL, INTENT(IN)` for `vssqrt`
`DOUBLE PRECISION, INTENT(IN)` for `vdsqrt`
Array, specifies the input vector a.

C:

n `int`. Specifies the number of elements to be calculated.

a `const float*` for `vsSqrt`
`const double*` for `vdSqrt`
Pointer to an array that contains the input vector a.

Output Parameters

Fortran:

y `REAL` for `vssqrt`
`DOUBLE PRECISION` for `vdsqrt`
Array, specifies the output vector y.

C:

y `float*` for `vsSqrt`
`double*` for `vdSqrt`
Pointer to an array that contains the output vector y.

InvSqrt

Computes an inverse square root of vector elements.

Syntax

Fortran:

```
call vsinvsqrt( n, a, y )
call vdinvsqrt( n, a, y )
```

C:

```
vsInvSqrt( n, a, y );
vdInvSqrt( n, a, y );
```

Input Parameters

Fortran:

n — `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

a — `REAL, INTENT(IN)` for `vsinvsqrt`
`DOUBLE PRECISION, INTENT(IN)` for `vdinvsqrt`
Array, specifies the input vector a.

C:

n — `int`. Specifies the number of elements to be calculated.

a — `const float*` for `vsInvSqrt`
`const double*` for `vdInvSqrt`
Pointer to an array that contains the input vector a.

Output Parameters

Fortran:

y — `REAL` for `vsinvsqrt`
`DOUBLE PRECISION` for `vdinvsqrt`
Array, specifies the output vector y.

C:

y — `float*` for `vsInvSqrt`
`double*` for `vdInvSqrt`
Pointer to an array that contains the output vector y.

Cbrt

Computes a cube root of vector elements.

Syntax

Fortran:

```
call vscbrt( n, a, y )
call vdcbrt( n, a, y )
```

C:

```
vsCbrt( n, a, y );
vdCbrt( n, a, y );
```

Input Parameters

Fortran:

n — `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

a — `REAL, INTENT(IN)` for `vscbrt`
`DOUBLE PRECISION, INTENT(IN)` for `vdcbrt`
Array, specifies the input vector a.

C:

n — `int`. Specifies the number of elements to be calculated.

a — `const float*` for `vsCbrt`
`const double*` for `vdCbrt`
Pointer to an array that contains the input vector a.

Output Parameters

Fortran:

y — `REAL` for `vscbrt`
`DOUBLE PRECISION` for `vdcbrt`
Array, specifies the output vector y.

C:

y `float*` for `vsCbrt`
`double*` for `vdCbrt`
Pointer to an array that contains the output vector `y`.

InvCbrt

Computes an inverse cube root of vector elements.

Syntax

Fortran:

```
call vsinvcbrt( n, a, y )
call vdinvcbrt( n, a, y )
```

C:

```
vsInvCbrt( n, a, y );
vdInvCbrt( n, a, y );
```

Input Parameters

Fortran:

n `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

a `REAL, INTENT(IN)` for `vsinvcbrt`
`DOUBLE PRECISION, INTENT(IN)` for `vdinvcbrt`
Array, specifies the input vector `a`.

C:

n `int`. Specifies the number of elements to be calculated.

a `const float*` for `vsInvCbrt`
`const double*` for `vdInvCbrt`
Pointer to an array that contains the input vector `a`.

Output Parameters

Fortran:

y — `REAL` for `vsinvcbrt`
`DOUBLE PRECISION` for `vdinvcbrt`
Array, specifies the output vector y.

C:

y — `float*` for `vsInvCbrt`
`double*` for `vdInvCbrt`
Pointer to an array that contains the output vector y.

Pow

Computes a *to the power* b
for elements of two vectors.

Syntax

Fortran:

```
call vspow( n, a, b, y )
call vdpow( n, a, b, y )
```

C:

```
vsPow( n, a, b, y );
vdPow( n, a, b, y );
```

Input Parameters

Fortran:

n — `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

a, *b* — `REAL, INTENT(IN)` for `vspow`
`DOUBLE PRECISION, INTENT(IN)` for `vdpow`
Arrays, specify the input vectors a and b.

C:

n `int`. Specifies the number of elements to be calculated.

a, *b* `const float*` for `vsPow`
`const double*` for `vdPow`
Pointers to arrays that contain the input vectors `a` and `b`.

Table 9-4 Precision Overflow Thresholds for Pow Function

Data Type	Threshold Limitations on Input Parameters
single precision	`abs(a[i]) < ( FLT_MAX )` $^{1/b[i]}$
double precision	`abs(a[i]) < ( DBL_MAX )` $^{1/b[i]}$

Output Parameters

Fortran:

y `REAL` for `vspow`
`DOUBLE PRECISION` for `vdpow`
Array, specifies the output vector `y`.

C:

y `float*` for `vsPow`
`double*` for `vdPow`
Pointer to an array that contains the output vector `y`.

Description

The function `Pow` has certain limitations on the input range of *a* and *b* parameters. Specifically, if *a*`[i]` is positive, then *b*`[i]` may be arbitrary. For negative or zero *a*`[i]`, the value of *b*`[i]` must be integer (either positive or negative).

Powx

Raises each element of a vector to the constant power.

Syntax

Fortran:

```
call vspowx( n, a, b, y )
call vdpowx( n, a, b, y )
```

C:

```
vsPowx( n, a, b, y );
vdPowx( n, a, b, y );
```

Input Parameters

Fortran:

n	`INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.
a, *b*	`REAL, INTENT(IN)` for `vspowx` `DOUBLE PRECISION, INTENT(IN)` for `vdpowx` Array *a* specifies the input vector; scalar value *b* is the constant power.

C:

n	`int`. Specifies the number of elements to be calculated.
a	`const float*` for `vsPowx` `const double*` for `vdPowx` Pointer to an array that contains the input vector a.
b	`const float` for `vsPowx` `const double` for `vdPowx` Constant value for power b.

Table 9-5 Precision Overflow Thresholds for Powx Function

Data Type	Threshold Limitations on Input Parameters
single precision	abs(*a*[i]) < (FLT_MAX) $^{1/b}$
double precision	abs(*a*[i]) < (DBL_MAX) $^{1/b}$

Output Parameters

Fortran:

y — `REAL` for `vspowx`
`DOUBLE PRECISION` for `vdpowx`
Array, specifies the output vector y.

C:

y — `float*` for `vsPowx`
`double*` for `vdPowx`
Pointer to an array that contains the output vector y.

Description

The function `Powx` has certain limitations on the input range of *a* and *b* parameters. Specifically, if *a*[i] is positive, then *b* may be arbitrary. For negative or zero *a*[i], the value of *b* must be integer (either positive or negative).

Exp

Computes an exponential of vector elements.

Syntax

Fortran:

```
call vsexp( n, a, y )
call vdexp( n, a, y )
```

C:

```
vsExp( n, a, y );
vdExp( n, a, y );
```

Input Parameters

Fortran:

n	`INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.
a	`REAL, INTENT(IN)` for `vsexp` `DOUBLE PRECISION, INTENT(IN)` for `vdexp` Array, specifies the input vector `a`.

C:

n	`int`. Specifies the number of elements to be calculated.
a	`const float*` for `vsExp` `const double*` for `vdExp` Pointer to an array that contains the input vector `a`.

Table 9-6 Precision Overflow Thresholds for Exp Function

Data Type	Threshold Limitations on Input Parameters
single precision	`a[i] < Ln( FLT_MAX )`
double precision	`a[i] < Ln( DBL_MAX )`

Output Parameters

Fortran:

y	`REAL` for `vsexp` `DOUBLE PRECISION` for `vdexp` Array, specifies the output vector `y`.

C:

y	`float*` for `vsExp` `double*` for `vdExp` Pointer to an array that contains the output vector `y`.

Ln

Computes natural logarithm of vector elements.

Syntax

Fortran:

```
call vsln( n, a, y )
call vdln( n, a, y )
```

C:

```
vsLn( n, a, y );
vdLn( n, a, y );
```

Input Parameters

Fortran:

n	`INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.
a	`REAL, INTENT(IN)` for `vsln` `DOUBLE PRECISION, INTENT(IN)` for `vdln` Array, specifies the input vector `a`.

C:

n	`int`. Specifies the number of elements to be calculated.
a	`const float*` for `vsLn` `const double*` for `vdLn` Pointer to an array that contains the input vector `a`.

Output Parameters

Fortran:

y	`REAL` for `vsln` `DOUBLE PRECISION` for `vdln` Array, specifies the output vector `y`.

C:

y	`float*` for `vsLn` `double*` for `vdLn` Pointer to an array that contains the output vector `y`.

Log10

Computes denary logarithm of vector elements.

Syntax

Fortran:

```
call vslog10( n, a, y )
call vdlog10( n, a, y )
```

C:

```
vsLog10( n, a, y );
vdLog10( n, a, y );
```

Input Parameters

Fortran:

n	`INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.
a	`REAL, INTENT(IN)` for `vslog10` `DOUBLE PRECISION, INTENT(IN)` for `vdlog10` Array, specifies the input vector `a`.

C:

n	`int`. Specifies the number of elements to be calculated.
a	`const float*` for `vsLog10` `const double*` for `vdLog10` Pointer to an array that contains the input vector `a`.

Output Parameters

Fortran:

y `REAL` for `vslog10`
`DOUBLE PRECISION` for `vdlog10`
Array, specifies the output vector `y`.

C:

y `float*` for `vsLog10`
`double*` for `vdLog10`
Pointer to an array that contains the output vector `y`.

Cos

Computes cosine of vector elements.

Syntax

Fortran:

```
call vscos( n, a, y )
call vdcos( n, a, y )
```

C:

```
vsCos( n, a, y );
vdCos( n, a, y );
```

Input Parameters

Fortran:

n `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

a `REAL, INTENT(IN)` for `vscos`
`DOUBLE PRECISION, INTENT(IN)` for `vdcos`
Array, specifies the input vector `a`.

C:

n	`int`. Specifies the number of elements to be calculated.
a	`const float*` for `vsCos` `const double*` for `vdCos` Pointer to an array that contains the input vector a.

Output Parameters

Fortran:

y	`REAL` for `vscos` `DOUBLE PRECISION` for `vdcos` Array, specifies the output vector y.

C:

y	`float*` for `vsCos` `double*` for `vdCos` Pointer to an array that contains the output vector y.

Sin

Computes sine of vector elements.

Syntax

Fortran:

```
call vssin( n, a, y )
call vdsin( n, a, y )
```

C:

```
vsSin( n, a, y );
vdSin( n, a, y );
```

Input Parameters

Fortran:

n	`INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.
a	`REAL, INTENT(IN)` for `vssin` `DOUBLE PRECISION, INTENT(IN)` for `vdsin` Array, specifies the input vector `a`.

C:

n	`int`. Specifies the number of elements to be calculated.
a	`const float*` for `vsSin` `const double*` for `vdSin` Pointer to an array that contains the input vector `a`.

Output Parameters

Fortran:

y	`REAL` for `vssin` `DOUBLE PRECISION` for `vdsin` Array, specifies the output vector `y`.

C:

y	`float*` for `vsSin` `double*` for `vdSin` Pointer to an array that contains the output vector `y`.

SinCos

Computes sine and cosine of vector elements.

Syntax

Fortran:

```
call vssincos( n, a, y, z )
call vdsincos( n, a, y, z )
```

C:

```
vsSinCos( n, a, y, z );
vdSinCos( n, a, y, z );
```

Input Parameters

Fortran:

n	`INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.
a	`REAL, INTENT(IN)` for `vssincos` `DOUBLE PRECISION, INTENT(IN)` for `vdsincos` Array, specifies the input vector `a`.

C:

n	`int`. Specifies the number of elements to be calculated.
a	`const float*` for `vsSinCos` `const double*` for `vdSinCos` Pointer to an array that contains the input vector `a`.

Output Parameters

Fortran:

y, *z*	`REAL` for `vssincos` `DUBLE PRECISION` for `vdsincos` Arrays, specify the output vectors `y` (for sine values) and *z* (for cosine values).

C:

y, *z*	`float*` for `vsSinCos` `double*` for `vdSinCos` Pointers to arrays that contain the output vectors `y` (for sine values) and *z* (for cosine values).

Tan

Computes tangent of vector elements.

Syntax

Fortran:

```
call vstan( n, a, y )
call vdtan( n, a, y )
```

C:

```
vsTan( n, a, y );
vdTan( n, a, y );
```

Input Parameters

Fortran:

n — `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

a — `REAL, INTENT(IN)` for `vstan`
`DOUBLE PRECISION, INTENT(IN)` for `vdtan`
Array, specifies the input vector a.

C:

n — `int`. Specifies the number of elements to be calculated.

a — `const float*` for `vsTan`
`const double*` for `vdTan`
Pointer to an array that contains the input vector a.

Output Parameters

Fortran:

y — `REAL` for `vstan`
`DOUBLE PRECISION` for `vdtan`
Array, specifies the output vector y.

C:

y — `float*` for `vsTan`
`double*` for `vdTan`
Pointer to an array that contains the output vector y.

Acos

Computes inverse cosine of vector elements.

Syntax

Fortran:

```
call vsacos( n, a, y )
call vdacos( n, a, y )
```

C:

```
vsAcos( n, a, y );
vdAcos( n, a, y );
```

Input Parameters

Fortran:

n — `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

a — `REAL, INTENT(IN)` for `vsacos`
`DOUBLE PRECISION, INTENT(IN)` for `vdacos`
Array, specifies the input vector a.

C:

n — `int`. Specifies the number of elements to be calculated.

a — `const float*` for `vsAcos`
`const double*` for `vdAcos`
Pointer to an array that contains the input vector a.

Output Parameters

Fortran:

y — `REAL` for `vsacos`
`DOUBLE PRECISION` for `vdacos`
Array, specifies the output vector `y`.

C:

y — `float*` for `vsAcos`
`double*` for `vdAcos`
Pointer to an array that contains the output vector `y`.

Asin

Computes inverse sine of vector elements.

Syntax

Fortran:

```
call vsasin( n, a, y )
call vdasin( n, a, y )
```

C:

```
vsAsin( n, a, y );
vdAsin( n, a, y );
```

Input Parameters

Fortran:

n — `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

a — `REAL, INTENT(IN)` for `vsasin`
`DOUBLE PRECISION, INTENT(IN)` for `vdasin`
Array, specifies the input vector `a`.

C:

n `int`. Specifies the number of elements to be calculated.

a `const float*` for `vsAsin`
`const double*` for `vdAsin`
Pointer to an array that contains the input vector a.

Output Parameters

Fortran:

y `REAL` for `vsasin`
`DOUBLE PRECISION` for `vdasin`
Array, specifies the output vector y.

C:

y `float*` for `vsAsin`
`double*` for `vdAsin`
Pointer to an array that contains the output vector y.

Atan

Computes inverse tangent of vector elements.

Syntax

Fortran:

```
call vsatan( n, a, y )
call vdatan( n, a, y )
```

C:

```
vsAtan( n, a, y );
vdAtan( n, a, y );
```

Input Parameters

Fortran:

n `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

a `REAL, INTENT(IN)` for `vsatan`
`DOUBLE PRECISION, INTENT(IN)` for `vdatan`
Array, specifies the input vector `a`.

C:

n `int`. Specifies the number of elements to be calculated.

a `const float*` for `vsAtan`
`const double*` for `vdAsin`
Pointer to an array that contains the input vector `a`.

Output Parameters

Fortran:

y `REAL` for `vsatan`
`DOUBLE PRECISION` for `vdatan`
Array, specifies the output vector `y`.

C:

y `float*` for `vsAtan`
`double*` for `vdAtan`
Pointer to an array that contains the output vector `y`.

Atan2

Computes four-quadrant inverse tangent of elements of two vectors.

Syntax

Fortran:

```
call vsatan2( n, a, b, y )
call vdatan2( n, a, b, y )
```

C:

```
vsAtan2( n, a, b, y );
vdAtan2( n, a, b, y );
```

Input Parameters

Fortran:

n — `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

a, *b* — `REAL, INTENT(IN)` for `vsatan2`
`DOUBLE PRECISION, INTENT(IN)` for `vdatan2`
Arrays, specify the input vectors `a` and `b`.

C:

n — `int`. Specifies the number of elements to be calculated.

a, *b* — `const float*` for `vsAtan2`
`const double*` for `vdAtan2`
Pointers to arrays that contain the input vectors `a` and `b`.

Output Parameters

Fortran:

y — `REAL` for `vsatan2`
`DOUBLE PRECISION` for `vdatan2`
Array, specifies the output vector `y`.

C:

y — `float*` for `vsAtan2`
`double*` for `vdAtan2`
Pointer to an array that contains the output vector `y`.

The elements of the output vector `y` are computed as the four-quadrant arctangent of *a*`[i] /` *b*`[i]`.

Cosh

Computes hyperbolic cosine of vector elements.

Syntax

Fortran:

```
call vscosh( n, a, y )
call vdcosh( n, a, y )
```

C:

```
vsCosh( n, a, y );
vdCosh( n, a, y );
```

Input Parameters

Fortran:

n	`INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.
a	`REAL, INTENT(IN)` for `vscosh` `DOUBLE PRECISION, INTENT(IN)` for `vdcosh` Array, specifies the input vector `a`.

C:

n	`int`. Specifies the number of elements to be calculated.
a	`const float*` for `vsCosh` `const double*` for `vdCosh` Pointer to an array that contains the input vector `a`.

Table 9-7 Precision Overflow Thresholds for Cosh Function

Data Type	Threshold Limitations on Input Parameters
single precision	`-Ln(FLT_MAX)-Ln2 < a[i] < Ln(FLT_MAX)+Ln2`
double precision	`-Ln(DBL_MAX)-Ln2 < a[i] < Ln(DBL_MAX)+Ln2`

Output Parameters

Fortran:

y — `REAL` for `vscosh`
`DOUBLE PRECISION` for `vdcosh`
Array, specifies the output vector y.

C:

y — `float*` for `vsCosh`
`double*` for `vdCosh`
Pointer to an array that contains the output vector y.

Sinh

Computes hyperbolic sine of vector elements.

Syntax

Fortran:

```
call vssinh( n, a, y )
call vdsinh( n, a, y )
```

C:

```
vsSinh( n, a, y );
vdSinh( n, a, y );
```

Input Parameters

Fortran:

n — `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

a `REAL, INTENT(IN)` for `vssinh`
`DOUBLE PRECISION, INTENT(IN)` for `vdsinh`
Array, specifies the input vector `a`.

C:

n `int`. Specifies the number of elements to be calculated.

a `const float*` for `vsSinh`
`const double*` for `vdSinh`
Pointer to an array that contains the input vector `a`.

Table 9-8 Precision Overflow Thresholds for `Sinh` Function

Data Type	Threshold Limitations on Input Parameters
single precision	`-Ln(FLT_MAX)-Ln2 < a[i] < Ln(FLT_MAX)+Ln2`
double precision	`-Ln(DBL_MAX)-Ln2 < a[i] < Ln(DBL_MAX)+Ln2`

Output Parameters

Fortran:

y `REAL` for `vssinh`
`DOUBLE PRECISION` for `vdsinh`
Array, specifies the output vector `y`.

C:

y `float*` for `vsSinh`
`double*` for `vdSinh`
Pointer to an array that contains the output vector `y`.

Tanh

Computes hyperbolic tangent of vector elements.

Syntax

Fortran:

```
call vstanh( n, a, y )
call vdtanh( n, a, y )
```

C:

```
vsTanh( n, a, y );
vdTanh( n, a, y );
```

Input Parameters

Fortran:

n	`INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.
a	`REAL, INTENT(IN)` for `vstanh` `DOUBLE PRECISION, INTENT(IN)` for `vdtanh` Array, specifies the input vector a.

C:

n	`int`. Specifies the number of elements to be calculated.
a	`const float*` for `vsTanh` `const double*` for `vdTanh` Pointer to an array that contains the input vector a.

Output Parameters

Fortran:

y	`REAL` for `vstanh` `DOUBLE PRECISION` for `vdtanh` Array, specifies the output vector y.

C:

y float* for vsTanh
double* for vdTanh
Pointer to an array that contains the output vector y.

Acosh

Computes inverse hyperbolic cosine (nonnegative) of vector elements.

Syntax

Fortran:

```
call vsacosh( n, a, y )
call vdacosh( n, a, y )
```

C:

```
vsAcosh( n, a, y );
vdAcosh( n, a, y );
```

Input Parameters

Fortran:

n INTEGER, INTENT(IN). Specifies the number of elements to be calculated.

a REAL, INTENT(IN) for vsacosh
DOUBLE PRECISION, INTENT(IN) for vdacosh
Array, specifies the input vector a.

C:

n int. Specifies the number of elements to be calculated.

a const float* for vsAcosh
const double* for vdAcosh
Pointer to an array that contains the input vector a.

Output Parameters

Fortran:

y — `REAL` for `vsacosh`
`DOUBLE PRECISION` for `vdacosh`
Array, specifies the output vector `y`.

C:

y — `float*` for `vsAcosh`
`double*` for `vdAcosh`
Pointer to an array that contains the output vector `y`.

Asinh

Computes inverse hyperbolic sine of vector elements.

Syntax

Fortran:

```
call vsasinh( n, a, y )
call vdasinh( n, a, y )
```

C:

```
vsAsinh( n, a, y );
vdAsinh( n, a, y );
```

Input Parameters

Fortran:

n — `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

a — `REAL, INTENT(IN)` for `vsasinh`
`DOUBLE PRECISION, INTENT(IN)` for `vdasinh`
Array, specifies the input vector `a`.

C:

n `int`. Specifies the number of elements to be calculated.

a `const float*` for `vsAsinh`
`const double*` for `vdAsinh`
Pointer to an array that contains the input vector `a`.

Output Parameters

Fortran:

y `REAL` for `vsasinh`
`DOUBLE PRECISION` for `vdasinh`
Array, specifies the output vector `y`.

C:

y `float*` for `vsAsinh`
`double*` for `vdAsinh`
Pointer to an array that contains the output vector `y`.

Atanh

Computes inverse hyperbolic tangent of vector elements.

Syntax

Fortran:

```
call vsatanh( n, a, y )
call vdatanh( n, a, y )
```

C:

```
vsAtanh( n, a, y );
vdAtanh( n, a, y );
```

Input Parameters

Fortran:

n	`INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.
a	`REAL, INTENT(IN)` for `vsatanh` `DOUBLE PRECISION, INTENT(IN)` for `vdatanh` Array, specifies the input vector `a`.

C:

n	`int`. Specifies the number of elements to be calculated.
a	`const float*` for `vsAtanh` `const double*` for `vdAtanh` Pointer to an array that contains the input vector `a`.

Output Parameters

Fortran:

y	`REAL` for `vsatanh` `DOUBLE PRECISION` for `vdatanh` Array, specifies the output vector `y`.

C:

y	`float*` for `vsAtanh` `double*` for `vdAtanh` Pointer to an array that contains the output vector `y`.

Erf

Computes the error function value of vector elements.

Syntax

Fortran:

```
call vserf( n, a, y )
call vderf( n, a, y )
```

C:

```
vsErf( n, a, y );
vdErf( n, a, y );
```

Input Parameters

Fortran:

n — `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

a — `REAL, INTENT(IN)` for `vserf`
`DOUBLE PRECISION, INTENT(IN)` for `vderf`
Array, specifies the input vector `a`.

C:

n — `int`. Specifies the number of elements to be calculated.

a — `const float*` for `vsErf`
`const double*` for `vdErf`
Pointer to an array that contains the input vector `a`.

Output Parameters

Fortran:

y — `REAL` for `vserf`
`DOUBLE PRECISION` for `vderf`
Array, specifies the output vector `y`.

C:

y — `float*` for `vsErf`
`double*` for `vdErf`
Pointer to an array that contains the output vector `y`.

Description

The function `Erf` computes the error function values for elements of the input vector *a* and writes them to the output vector *y*.

The error function is defined as given by:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$$

Erfc

Computes the complementary error function value of vector elements.

Syntax

Fortran:

```
call vserfc( n, a, y )
call vderfc( n, a, y )
```

C:

```
vsErfc( n, a, y );
vdErfc( n, a, y );
```

Input Parameters

Fortran:

n — `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

a — `REAL, INTENT(IN)` for `vserfc`
`DOUBLE PRECISION, INTENT(IN)` for `vderfc`
Array, specifies the input vector a.

C:

n — `int`. Specifies the number of elements to be calculated.

a — `const float*` for `vsErfc`
`const double*` for `vdErfc`
Pointer to an array that contains the input vector a.

Output Parameters

Fortran:

y	`REAL` for `vserfc` `DOUBLE PRECISION` for `vderfc` Array, specifies the output vector `y`.

C:

y	`float*` for `vsErfc` `double*` for `vdErfc` Pointer to an array that contains the output vector `y`.

Description

The function `Erfc` computes the error function values for elements of the input vector *a* and writes them to the output vector *y*.

The error function is defined as given by:

$$\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$$

or, in other words,

$$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}}\int_{x}^{\infty} e^{-t^2}dt .$$

VML Pack/Unpack Functions

This section describes VML functions which convert vectors with unit increment to and from vectors with positive increment indexing, vector indexing and mask indexing (see Appendix B for details on vector indexing methods).

Table 9-9 lists available VML Pack/Unpack functions, together with data types and indexing methods associated with them.

Table 9-9 VML Pack/Unpack Functions

Function Short Name	Data Types	Indexing Methods	Description
`Pack`	`s, d`	`I,V,M`	Gathers elements of arrays, indexed by different methods.
`Unpack`	`s, d`	`I,V,M`	Scatters vector elements to arrays with different indexing.

Pack

Copies elements of an array with specified indexing to a vector with unit increment.

Syntax

Fortran:

```
call vspacki( n, a, inca, y  )
call vspackv( n, a, ia,   y  )
call vspackm( n, a, ma,   y  )
call vdpacki( n, a, inca, y )
call vdpackv( n, a, ia,   y )
call vdpackm( n, a, ma,   y )
```

C:

```
vsPackI( n, a, inca, y );
vsPackV( n, a, ia,   y );
vsPackM( n, a, ma,   y );
vdPackI( n, a, inca, y );
vdPackV( n, a, ia,    y );
vdPackM( n, a, ma,    y );
```

Input Parameters

Fortran:

n — `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

a — `REAL, INTENT(IN)` for `vspacki`, `vspackv`, `vspackm`
`DOUBLE PRECISION, INTENT(IN)` for `vdpacki`, `vdpackv`, `vdpackm`
Array, `DIMENSION` at least `(1 + (n-1)*inca)` for `vspacki`,
at least `max( n,max(ia[j]) ),j=0,…,n-1,` for `vspackv`,
at least *n* for `vspackm`
Specifies the input vector a.

inca — `INTEGER, INTENT(IN)` for `vspacki`, `vdpacki`.
Specifies the increment for the elements of *a*.

ia — `INTEGER, INTENT(IN)` for `vspackv`, `vdpackv`.
Array, `DIMENSION` at least *n*
Specifies the index vector for the elements of *a*.

ma — `INTEGER, INTENT(IN)` for `vspackm`, `vdpackm`.
Array, `DIMENSION` at least *n*
Specifies the mask vector for the elements of *a*.

C:

n — `int`. Specifies the number of elements to be calculated

a — `const float*` for `vsPackI`, `vsPackV`, `vsPackM`
`const double*` for `vdPackI`, `vdPackV`, `vdPackM`
Specifies the pointer to an array that contains the input vector a.
Size of the array must be:
at least `(1 + (n-1)*inca)` for `vsPackI,`
at least `max( n,max(ia[j]) ),j=0,…,n-1,` for `vsPackV,`
at least *n* for `vsPackM.`

inca — `int` for `vsPackI`, `vdPackI`.
Specifies the increment for the elements of *a*.

ia — `const int*` for `vsPackV`, `vdPackV`. Specifies the pointer to an array of size at least *n* that contains the index vector for the elements of *a*.

ma — `const int*` for `vsPackM`, `vdPackM`. Specifies the pointer to an array of size at least *n* that contains the mask vector for the elements of *a*.

Output Parameters

Fortran:

y — `REAL` for `vspacki`, `vspackv`, `vspackm`
`DOUBLE PRECISION` for `vdpacki`, `vdpackv`, `vdpackm`
Array, `DIMENSION` at least *n*, specifies the output vector y.

C:

y — `float*` for `vsPackI`, `vsPackV`, `vsPackM`
`double*` for `vdPackI`, `vdPackV`, `vdPackM`
Specifies the pointer to an array of size at least *n* that contains the output vector y.

Unpack

Copies elements of a vector with unit increment to an array with specified indexing.

Syntax

Fortran:

```
call vsunpacki( n, a, y, incy )
call vsunpackv( n, a, y, iy )
call vsunpackm( n, a, y, my )
call vdunpacki( n, a, y, incy )
call vdunpackv( n, a, y, iy )
call vdunpackm( n, a, y, my )
```

C:

```
vsUnpackI( n, a, y, incy );
vsUnpackV( n, a, y, iy );
vsUnpackM( n, a, y, my );
vdUnpackI( n, a, y, incy );
vdUnpackV( n, a, y, iy );
vdUnpackM( n, a, y, my );
```

Input Parameters

Fortran:

n `INTEGER, INTENT(IN)`. Specifies the number of elements to be calculated.

a `REAL, INTENT(IN)` for `vsunpacki`, `vsunpackv`, `vsunpackm`
`DOUBLE PRECISION, INTENT(IN)` for `vdunpacki`, `vdunpackv`, `vdunpackm`.
Array, `DIMENSION` at least *n*, specifies the input vector `a`.

incy `INTEGER, INTENT(IN)` for `vsunpacki`, `vdunpacki`.
Specifies the increment for the elements of *y*.

iy `INTEGER, INTENT(IN)` for `vsunpackv`, `vdunpackv`.
Array, `DIMENSION` at least *n*, specifies the index vector for the elements of *y*.

my `INTEGER, INTENT(IN)` for `vsunpackm`, `vdunpackm`.
Array, `DIMENSION` at least *n*, specifies the mask vector for the elements of *y*.

C:

n `int`. Specifies the number of elements to be calculated.

a `const float*` for `vsUnpackI`, `vsUnpackV`, `vsUnpackM`
`const double*` for `vdUnpackI`, `vdUnpackV`, `vdUnpackM`
Specifies the pointer to an array of size at least *n* that contains the input vector `a`.

incy `int` for `vsUnpackI`, `vdUnpackI`.
Specifies the increment for the elements of *y*.

iy `const int*` for `vsUnpackV`, `vdUnpackV`. Specifies the pointer to an array of size at least *n* that contains the index vector for the elements of *a*.

my `const int*` for `vsUnpackM`, `vdUnpackM`. Specifies the pointer to an array of size at least *n* that contains the mask vector for the elements of *a*.

Output Parameters

Fortran:

y	`REAL` for `vsunpacki`, `vsunpackv`, `vsunpackm` `DOUBLE PRECISION` for `vdunpacki`, `vdunpackv`, `vdunpackm`. Array, `DIMENSION` at least `(1 + (`*n*`-1)*`*incy*`)` for `vsunpacki`, at least `max( `*n*`,max(`*iy*`[j]) ),j=0,…,`*n*`-1,` for `vsunpackv`, at least *n* for `vsunpackm` Specifies the output vector y.

C:

y	`float*` for `vsUnpackI`, `vsUnpackV`, `vsUnpackM` `double*` for `vdUnpackI`, `vdUnpackV`, `vdUnpackM` Specifies the pointer to an array that contains the output vector y. Size of the array must be: at least `(1 + (`*n*`-1)*`*incy*`)` for `vsUnPackI`, at least `max( `*n*`,max(`*ia*`[j]) ),j=0,…,`*n*`-1,` for `vsUnPackV`, at least *n* for `vsUnPackM`.

VML Service Functions

This section describes VML functions which allow the user to set /get the accuracy mode, and set/get the error code. All these functions are available both in Fortran- and C- interfaces. Table 9-10 lists available VML Service functions and their short description.

Table 9-10 VML Service Functions

Function Short Name	Description
`SetMode`	Sets the VML mode
`GetMode`	Gets the VML mode
`“SetErrStatus”`	Sets the VML error status
`GetErrStatus`	Gets the VML error status
`ClearErrStatus`	Clears the VML error status
`SetErrorCallBack`	Sets the additional error handler callback function
`GetErrorCallBack`	Gets the additional error handler callback function

Table 9-10 VML Service Functions (continued)

Function Short Name	Description
`ClearErrorCallBack`	Deletes the additional error handler callback function

SetMode

Sets a new mode for VML functions according to `mode` *parameter and stores the previous VML mode to* `oldmode`.

Syntax

Fortran:

```
oldmode = vmlsetmode( mode )
```

C:

```
oldmode = vmlSetMode( mode );
```

Input Parameters

Fortran:

mode `INTEGER, INTENT(IN)`. Specifies the VML mode to be set.

C:

mode `int`. Specifies the VML mode to be set.

Output Parameters

Fortran:

oldmode `INTEGER`. Specifies the former VML mode.

C:

oldmode `int`. Specifies the former VML mode.

Description

The *mode* parameter is designed to control accuracy, FPU and error handling options. Table 9-11 lists values of the *mode* parameter. All other possible values of the *mode* parameter may be obtained from these values by using bitwise OR (|) operation to combine one value for accuracy, one for FPU, and one for error control options. The default value of the *mode* parameter is `VML_HA | VML_ERRMODE_DEFAULT`. Thus, the current FPU control word (FPU precision and the rounding method) is used by default.

If any VML mathematical function requires different FPU precision, or rounding method, it changes these options automatically and then restores the former values. The *mode* parameter enables you to minimize switching the internal FPU mode inside each VML mathematical function that works with similar precision and accuracy settings. To accomplish this, set the *mode* parameter to `VML_FLOAT_CONSISTENT` for single precision functions, or to `VML_DOUBLE_CONSISTENT` for double precision functions. These values of the *mode* parameter are the optimal choice for the respective function groups, as they are required for most of the VML mathematical functions. After the execution is over, set the *mode* to `VML_RESTORE` if you need to restore the previous FPU mode.

Table 9-11 Values of the *mode* Parameter

Value of *mode*	Description
Accuracy Control	
VML_HA	High accuracy versions of VML functions will be used
VML_LA	Low accuracy versions of VML functions will be used
Additional FPU Mode Control	
VML_FLOAT_CONSISTENT	The optimal FPU mode (control word) for single precision functions is set, and the previous FPU mode is saved
VML_DOUBLE_CONSISTENT	The optimal FPU mode (control word) for double precision functions is set, and the previous FPU mode is saved
VML_RESTORE	The previously saved FPU mode is restored
Error Mode Control	
VML_ERRMODE_IGNORE	No action is set for computation errors
VML_ERRMODE_ERRNO	On error, the `errno` variable is set
VML_ERRMODE_STDERR	On error, the error text information is written to `stderr`

Table 9-11 Values of the *mode* Parameter (continued)

Value of *mode*	Description
`VML_ERRMODE_EXCEPT`	On error, an exception is raised
`VML_ERRMODE_CALLBACK`	On error, an additional error handler function is called
`VML_ERRMODE_DEFAULT`	On error, the `errno` variable is set, an exception is raised, and an additional error handler function is called

Examples

Several examples of calling the function `vmlSetMode()` with different values of the *mode* parameter are given below:

Fortran:

```
oldmode = vmlsetmode( VML_LA )
call vmlsetmode( IOR(VML_LA, IOR(VML_FLOAT_CONSISTENT,
      VML_ERRMODE_IGNORE )))
call vmlsetmode( VML_RESTORE)
```

C:

```
vmlSetMode( VML_LA );
vmlSetMode( VML_LA | VML_FLOAT_CONSISTENT | VML_ERRMODE_IGNORE );
vmlSetMode( VML_RESTORE);
```

GetMode

Gets the VML mode.

Syntax

Fortran:

```
mod  = vmlgetmode()
```

C:

```
mod  = vmlGetMode( void );
```

Output Parameters

Fortran:

mod `INTEGER`. Specifies the packed *mode* parameter.

C:

mod `int`. Specifies the packed *mode* parameter.

Description

The function `vmlGetMode()` returns the VML *mode* parameter which controls accuracy, FPU and error handling options. The *mod* variable value is some combination of the values listed in the Table 9-11. You can obtain some of these values using the respective mask from the Table 9-12, for example:

Fortran:

```
mod  = vmlgetmode()
accm = IAND(mod, VML_ACCURACY_MASK)
fpum = IAND(mod, VML_FPUMODE_MASK)
errm = IAND(mod, VML_ERRMODE_MASK)
```

C:

```
accm = vmlGetMode(void )& VML_ACCURACY_MASK;
fpum = vmlGetMode(void )& VML_FPUMODE _MASK;
errm = vmlGetMode(void )& VML_ERRMODE _MASK;
```

Table 9-12 Values of Mask for the *mode* Parameter

Value of mask	Description
VML_ACCURACY_MASK	Specifies mask for accuracy *mode* selection.
VML_FPUMODE_MASK	Specifies mask for FPU *mode* selection.
VML_ERRMODE_MASK	Specifies mask for error *mode* selection.

SetErrStatus

Sets the new VML error status according to `err` and stores the previous VML error status to `olderr`.

Syntax

Fortran:

`olderr = vmlseterrstatus( err )`

C:

`olderr = vmlSetErrStatus( err );`

Input Parameters

Fortran:

err	`INTEGER, INTENT(IN)`. Specifies the VML error status to be set.

C:

err	`int`. Specifies the VML error status to be set.

Output Parameters

Fortran:

olderr	`INTEGER`. Specifies the former VML error status.

C:

olderr	`int`. Specifies the former VML error status.

Table 9-13 lists possible values of the *err* parameter.

Table 9-13 Values of the VML Error Status

Error Status	Description
`VML_STATUS_OK`	The execution was completed successfully.
`VML_STATUS_BADSIZE`	The array dimension is not positive.
`VML_STATUS_BADMEM`	NULL pointer is passed.
`VML_STATUS_ERRDOM`	At least one of array values is out of a range of definition.
`VML_STATUS_SING`	At least one of array values caused a singularity.
`VML_STATUS_OVERFLOW`	An overflow has happened during the calculation process.
`VML_STATUS_UNDERFLOW`	An underflow has happened during the calculation process.

Examples:

```
vmlSetErrStatus( VML_STATUS_OK );
vmlSetErrStatus( VML_STATUS_ERRDOM );
vmlSetErrStatus( VML_STATUS_UNDERFLOW );
```

GetErrStatus

Gets the VML error status.

Syntax

Fortran:

```
err = vmlgeterrstatus( )
```

C:

```
err  = vmlGetErrStatus( void );
```

Output Parameters

Fortran:

err INTEGER. Specifies the VML error status.

C:

err `int`. Specifies the VML error status.

ClearErrStatus

Sets the VML error status to `VML_STATUS_OK` *and stores the previous VML error status to* `olderr`.

Syntax

Fortran:

```
olderr = vmlclearerrstatus( )
```

C:

```
olderr = vmlClearErrStatus( void );
```

Output Parameters

Fortran:

olderr `INTEGER`. Specifies the former VML error status.

C:

olderr `int`. Specifies the former VML error status.

SetErrorCallBack

Sets the additional error handler callback function and gets the old callback function.

Syntax

Fortran:

```
oldcallback = vmlseterrorcallback( callback )
```

C:

```
oldcallback = vmlSetErrorCallBack( callback );
```

Input Parameters

Fortran:

callback Address of the callback function.
The callback function has the following format:

```
INTEGER FUNCTION ERRFUNC(par)
 TYPE (ERROR_STRUCTURE) par
 ! ...
 ! user error processing
 ! ...
 ERRFUNC = 0
 ! if ERRFUNC  = 0 - standard VML error handler
             ! is called after the callback
 ! if ERRFUNC != 0 - standard VML error handler
             ! is not called
END
```

The passed error structure is defined as follows:

```
TYPE ERROR_STRUCTURE
                 SEQUENCE
     INTEGER*4 ICODE
     INTEGER*4 IINDEX
     REAL*8 DBA1
     REAL*8 DBA2
     REAL*8 DBR1
     REAL*8 DBR2
     CHARACTER(64) CFUNCNAME
     INTEGER*4 IFUNCNAMELEN
 END TYPE ERROR_STRUCTURE
```

C:

callback Pointer to the callback function.
The callback function has the following format:

```
static int __stdcall MyHandler(DefVmlErrorContext*
pContext)
```

```
{
    /* Handler body */
};
```

The passed error structure is defined as follows:

```
typedef struct _DefVmlErrorContext
{
    int iCode;        /* Error status value */
    int iIndex;     /* Index for bad array
            element, or bad array
            dimension, or bad
            array pointer */
    double dbA1;     * Error argument 1 */
    double dbA2;     /* Error argument 2 */
    double dbR1;     /* Error result 1 */
    double dbR2;     /* Error result 2 */
    char cFuncName[64]; /* Function name */
    int iFuncNameLen;  /* Length of function   name*/
} DefVmlErrorContext;
```

Output Parameters

Fortran:

`oldcallback` Address of the former callback function.

C:

`oldcallback` Pointer to the former callback function.

Description

The callback function is called on each VML mathematical function error if `VML_ERRMODE_CALLBACK` error mode is set (see Table 9-11).

Use the `vmlSetErrorCallBack()` function if you need to define your own callback function instead of default empty callback function.

The input structure for a callback function contains the following information about the encountered error:

- the input value which caused an error
- location (array index) of this value

- the computed result value
- error code
- name of the function in which the error occurred.

You can insert your own error processing into the callback function. This may include correcting the passed result values in order to pass them back and resume computation. The standard error handler is called after the callback function only if it returns `0`.

GetErrorCallBack

Gets the additional error handler callback function.

Syntax

Fortran:

```
fun = vmlgeterrorcallback( )
```

C:

```
fun = vmlGetErrorCallBack( void );
```

Output Parameters

Fortran:

fun	Address of the callback function.

C:

fun	Pointer to the callback function.

ClearErrorCallBack

Deletes the additional error handler callback function and retrieves the former callback function.

Syntax

Fortran:

oldcallback = vmlclearerrorcallback( )

C:

oldcallback = vmlClearErrorCallBack(void);

Output Parameters

Fortran:

oldcallback INTEGER. Address of the former callback function.

C:

oldcallback int. Pointer to the former callback function.

Vector Generators of Statistical Distributions 10

This chapter describes the part of Intel® MKL that is known as Vector Statistical Library (VSL) and is designed for the purpose of generating vectors of pseudorandom and quasi-random numbers.

VSL provides a set of subroutines implementing commonly used pseudo- or quasi-random number generators with continuous and discrete distribution. To speed up performance, all these subroutines were developed using the calls to the highly optimized *Basic Random Number Generators* (BRNGs) and the library of vector mathematical functions (VML, see Chapter 9, "Vector Mathematical Functions").

All VSL subroutines can be classified into three major categories:

- Transformation subroutines for different types of statistical distributions, for example, uniform, normal (Gaussian), binomial, etc. These subroutines indirectly call basic random number generators, which are either pseudorandom number generators or quasi-random number generators. Detailed description of the generators can be found in "Distribution Generators" section.
- Service subroutines to handle random number streams: create, initialize, delete, copy, get the index of a basic generator. The description of these subroutines can be found in "Service Subroutines" section.
- Registration subroutines for basic pseudorandom generators and subroutines that obtain properties of the registered generators (see "Advanced Service Subroutines" section).

The last two categories will be referred to as service subroutines.

Conventions

In this chapter no specific differentiation is made between random, pseudorandom, and quasi-random numbers, as well as between random, pseudorandom, and quasi-random number generators unless the context requires otherwise. For details, refer to '*Random Numbers'* section in *VSL Notes* document provided with Intel MKL.

All generators of nonuniform distributions, both discrete and continuous, are built on the basis of the uniform distribution generators, called Basic Random Number Generators (BRNGs). The pseudorandom numbers with nonuniform distribution are obtained through an appropriate transformation of the uniformly distributed pseudorandom numbers. Such transformations are referred to as *generation methods*. For a given distribution, several generation methods can be used. See *VSL Notes* for the description of methods available for each generator.

The *stream descriptor* specifies which BRNG should be used in a given transformation method. See '*Random Streams and RNGs in Parallel Computation*' secton of *VSL Notes*.

The term *computational node* means a logical or physical unit that can process data in parallel.

Mathematical Notation

The following notation is used throughout the text:

N	The set of natural numbers $N = \{1, 2, 3...\}$.
Z	The set of integers $Z = \{... -3, -2, -1, 0, 1, 2, 3 ...\}$.
R	The set of real numbers.
$\lfloor a \rfloor$	The floor of a (the largest integer less than or equal to a).
⊕ or **xor**	Bitwise exclusive OR.
C_α^k or $\binom{\alpha}{k}$	Binomial coefficient or combination ($\alpha \in R, \alpha \geq 0$; $k \in N \cup \{0\}$). $C_\alpha^0 = 1$. For $\alpha \geq k$ binomial coefficient is defined as $C_\alpha^k = \frac{\alpha(\alpha-1)\ldots(\alpha-k+1)}{k!}$. If $\alpha < k$, then $C_\alpha^k = 0$.

$\Phi(x)$ Cumulative Gaussian distribution function

$$\Phi(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{y^2}{2}\right) dy$$, defined over

$-\infty < x < +\infty$.

$\Phi(-\infty) = 0, \Phi(+\infty) = 1$.

LCG(a,c,m) Linear Congruential Generator $x_{n+1} = (ax_n + c) \bmod m$, where a is called the *multiplier*, c is called the *increment* and m is called the *modulus* of the generator.

MCG(a,m) Multiplicative Congruential Generator $x_{n+1} = (ax_n) \bmod m$ is a special case of Linear Congruential Generator, where the increment c is taken to be 0.

GFSR(p, q) Generalized Feedback Shift Register Generator $x_n = x_{n-p} \oplus x_{n-q}$.

Naming Conventions

The names of all VSL functions in FORTRAN are lowercase; names in C may contain both lowercase and uppercase letters.

The names of generator subroutines have the following structure:

`v<type of result>rng<distribution>` for FORTRAN-interface

`v<type of result>Rng<distribution>` for C-interface

where `v` is the prefix of a VSL vector function, and the field `<type of result>` is either `s` , `d` , or `i` and specifies one of the following types:

`s` `REAL` for FORTRAN-interface

`float` for C-interface

`d` `DOUBLE PRECISION` for FORTRAN-interface

`double` for C-interface

`i` INTEGER for FORTRAN-interface

`int` for C-interface

Prefixes `s` and `d` apply to continuous distributions only, prefix `i` applies only to discrete case. The prefix `rng` indicates that the subroutine is a random generator, and the `<distribution>` field specifies the type of statistical distribution.

Names of service subroutines follow the template below:

`vsl<name>` ,

where `vsl` is the prefix of a VSL service function. The field `<name>` contains a short function name. For a more detailed description of service subroutines refer to "Service Subroutines" and "Advanced Service Subroutines" sections.

Prototype of each generator subroutine corresponding to a given probability distribution fits the following structure:

`<function name>( method, stream, n, r, [<distribution parameters>] )`, where

- *method* is the number specifying the method of generation. A detailed description of this parameter can be found in "Distribution Generators" section.
- *stream* defines the random stream descriptor and must have a nonzero value. Random streams and their usage are discussed further in "Random Streams" and "Service Subroutines".
- *n* defines the number of random values to be generated. If *n* is less than or equal to zero, no values are generated. Furthermore, if *n* is negative, an error condition is set.
- *r* defines the destination array for the generated numbers. The dimension of the array must be large enough to store at least *n* random numbers.

Additional parameters included into `<distribution parameters>` field are individual for each generator subroutine and are described in detail in "Distribution Generators" section.

To invoke a distribution generator, use a call to the respective VSL subroutine. For example, to obtain a vector *r*, composed of *n* independent and identically distributed random numbers with normal (Gaussian) distribution, that have the mean value *a* and standard deviation *sigma*, write the following:

for FORTRAN-interface

```
call vsrnggaussian( method, stream, n, r, a, sigma )
```

for C-interface

```
vsRngGaussian( method, stream, n, r, a, sigma )
```

Basic Generators

VSL provides the following BRNGs, which differ in speed and other properties:

- the 32-bit multiplicative congruential pseudorandom number generator *MCG(1132489760,* 2^{31} *- 1)* [L'Ecuyer99],
- the 32-bit generalized feedback shift register pseudorandom number generator *GFSR(250,103)* [Kirkpatrick81],
- the combined multiple recursive pseudorandom number generator *MRG-32k3a* [L'Ecuyer99a],
- the 59-bit multiplicative congruential pseudorandom number generator *MCG(*13^{13}*,* 2^{59}*)*from NAG Numerical Libraries [NAG],
- Wichmann-Hill pseudorandom number generator (a set of 273 basic generators) from NAG Numerical Libraries [NAG].

Besides these pseudorandom number generators, VSL provides two basic quasi-random number generators:

- Sobol quasi-number generator [Sobol76], [Bratley88], which works in dimensions from 1 up to 40,
- Niederreiter quasi-random number generator [Bratley92], which works in dimensions from 1 up to 318.

Comparative performance analysis of the generators and some testing results can be found in *VSL Notes*.

VSL provides means of registration of such user-designed generators through the steps described in "Advanced Service Subroutines" section.
For some basic generators, VSL provides two methods of creating independent random streams in multiprocessor computations, which are the leapfrog method and the block-splitting method. Nevertheless, these sequence splitting methods are also useful in sequential Monte Carlo. In addition, Wichmann-Hill basic generator is a set of 273 pseudorandom number generators designed to create up to 273 independent random sequences, which might be used in parallel. The properties of the generators designed for parallel computations are discussed in detail in [Coddington94].
For a more detailed description of the generator properties and testing results refer to *VSL Notes*.

Random Streams

Random stream (or *stream*) is an abstract source of pseudo- and quasi-random sequences of uniform distribution. Users have no direct access to these sequences and operate with stream state descriptors only. A stream state descriptor, which holds state descriptive information for a particular BRNG, is a neccesary parameter in each subroutine of a distribution generator. Only subroutines of the distribution generator operate with random streams directly. See *VSL Notes* for details.

User can create unlimited number of random streams by VSL Service Subroutines like `NewStream` and utilize them in any distribution generator to get the sequence of numbers of given probability distribution. When they are no longer needed, the streams should be deleted calling service subroutine `DeleteStream`.

Data Types

FORTRAN:

```
TYPE VSL_STREAM_STATE
    INTEGER*4 descriptor1
    INTEGER*4 descriptor2
END TYPE VSL_STREAM_STATE
```

C:

```
typedef (void*) VSLStreamStatePtr;
```

See "Advanced Service Subroutines" for the format of the stream state structure for user-designed generators.

Service Subroutines

Stream handling comprises subroutines for creating, deleting, or copying the streams and getting the index of a basic generator.

Table 10-1 lists all available service subroutines

Table 10-1 Service Subroutines

Subroutine	Short Description
`NewStream`	Creates and initializes a random stream.
`NewStreamEx`	Creates and initializes a random stream for the generators with multiple initial conditions.

Table 10-1 Service Subroutines (continued)

Subroutine	Short Description
DeleteStream	Deletes previously created stream.
CopyStream	Copies a stream to another stream.
CopyStreamState	Creates a copy of a random stream state.
LeapfrogStream	Initializes the stream by the leapfrog method to generate a subsequnce of the original sequence.
SkipAheadStream	Initializes the stream by the skip-ahead method.
GetStreamStateBrng	Obtains the index of the basic generator responsible for the generation of a given random stream.
GetNumRegBrng	Obtains the number of currently registered basic generators.

NOTE. In the above table, the `vsl` prefix in the function names is omitted. In the function reference this prefix is always used in function prototypes and code examples.

Most of the generator-based work comprises three basic steps:

1. Creating and initializing a stream (`NewStream`, `NewStreamEx`, `CopyStream`, `CopyStreamState`, `LeapfrogStream`, `SkipAheadStream`).
2. Generating random numbers with given distribution, see "Distribution Generators".
3. Deleting the stream (`DeleteStream`).

Note that you can concurrently create multiple streams and obtain random data from one or several generators by using the stream state. You must use the `DeleteStream` function to delete all the streams afterwards.

NewStream

Creates and initializes a random stream.

Syntax

Fortran: c

```
all vslnewstream( stream, brng, seed )
```

C:

```
vslNewStream( stream, brng, seed )
```

Description

For a basic generator with number *brng* , this function creates a new stream and initializes it with a 32-bit seed. The seed is an initial value used to select a particular sequence generated by the basic generator *brng*. The function is also applicable for generators with multiple initial conditions. See *VSL Notes* for a more detailed description of stream initialization for different basic generators.

Input Parameters

FORTRAN:

brng	`INTEGER, INTENT(IN)`. Index of the basic generator to initialize the stream.
seed	`INTEGER, INTENT(IN)`. Initial condition of the stream. In the case of a quasi-random number generator *seed* parameter is used to set the dimension. If the dimension is greater than the dimension that *brng* can support or is less than 1, then the dimension is assumed to be equal to 1.

C:

brng	`int`. Index of the basic generator to initialize the stream.
seed	`unsigned int`. Initial condition of the stream.

Output Parameters

FORTRAN:

stream	`TYPE(VSL_STREAM_STATE), INTENT(OUT)`. Stream state descriptor.

C:

stream	`VSLStreamStatePtr*`. Pointer to the stream state structure.

NewStreamEx

Creates and initializes a random stream for generators with multiple initial conditions.

Syntax

Fortran:

```
call vslnewstreamex( stream, brng, n, params )
```

C:

```
vslNewStreamEx( stream, brng, n, params  )
```

Description

This function provides an advanced tool to set the initial conditions for a basic generator if its input arguments imply several initialization parameters. Initial values are used to select a particular sequence generated by the basic generator *brng*. Whenever possible, use `NewStream`, which is analogous to `vslNewStreamEx` except that it takes only one 32-bit initial condition. In particular, `vslNewStreamEx` may be used to initialize the state table in Generalized Feedback Shift Register Generators (GFSRs). A more detailed description of this issue can be found in *VSL Notes*.

Input Parameters

FORTRAN:

brng	`INTEGER, INTENT(IN).` Index of the basic generator to initialize the stream.
n	`INTEGER, INTENT(IN).` Number of initial conditions contained in *params*.
params	`INTEGER, INTENT(IN).` Array of initial conditions necessary for the basic generator *brng* to initialize the stream. In the case of a quasi-random number generator only the first element in *params* parameter is used to set the dimension. If the dimension is greater than the dimension that *brng* can support or is less than 1, then the dimension is assumed to be equal to 1.

C:

brng	`int`. Index of the basic generator to initialize the stream.
n	`int.` Number of initial conditions contained in *params*.
params	`const unsigned int[].` Array of initial conditions necessary for the basic generator *brng* to initialize the stream.

Output Parameters

FORTRAN:

stream	`TYPE(VSL_STREAM_STATE), INTENT(OUT).` Stream state descriptor.

C:

stream	`VSLStreamStatePtr*`. Pointer to the stream state structure.

DeleteStream

Deletes a random stream.

Syntax

Fortran:

`call vsldeletestream( stream )`

C:

`vslDeleteStream( stream )`

Description

This function deletes the random stream created by one of the initialization functions.

Input/Output Parameters

FORTRAN:

stream	`TYPE(VSL_STREAM_STATE), INTENT(INOUT)`. Descriptor of the stream to be deleted; must have non-zero value.

C:

stream	`VSLStreamStatePtr*`. Pointer to the stream state structure; must have non-zero value. After the stream is successfully deleted, the *stream* pointer is set to `NULL`.

CopyStream

Creates a copy of a random stream.

Syntax

Fortran:

`call vslcopystream( newstream, srcstream )`

C:

`vslCopyStream( newstream, srcstream )`

Description

The function creates an exact copy of *srcstream* and stores its descriptor to *newstream*.

Input Parameters

FORTRAN:

scrstream	`TYPE(VSL_STREAM_STATE), INTENT(IN)`. Descriptor of the stream to be copied.

C:

srcstream	`VSLStreamStatePtr`. Pointer to the stream state structure to be copied.

Output Parameters

FORTRAN:

newstream	`TYPE(VSL_STREAM_STATE), INTENT(OUT)`. Descriptor of the stream copy.

C:

newstream	`VSLStreamStatePtr*`. Pointer to the copy of the stream state structure.

CopyStreamState

Creates a copy of a random stream state.

Syntax

Fortran:

```
call vslcopystreamstate( deststream, srcstream )
```

C:

```
vslCopyStreamState( deststream, srcstream )
```

Description

The function copies a stream state from `srcstream` to the existing `deststream` stream. Both the streams should be generated by the same basic generator. En error message is generated when the index of the BRNG that produced `deststream` stream differs from the index of the BRNG that generated `srcstream` stream.

Unlike `CopyStream` function, which creates a new stream and copies both the stream state and other data from `srcstream`, the function `CopyStreamState` copies only `srcstream` stream state data to the generated `deststream` stream.

Input Parameters

FORTRAN:

scrstream	`TYPE(VSL_STREAM_STATE), INTENT(IN)`. Descriptor of the stream with the state to be copied.

C:

srcstream	`VSLStreamStatePtr`. Pointer to the stream state structure, from which the stream state is copied.

Output Parameters

FORTRAN:

deststream	`TYPE(VSL_STREAM_STATE), INTENT(IN)`. Descriptor of the destination stream where the state of *`scrstream`* stream is copied.

C:

deststream	`VSLStreamStatePtr`. Pointer to the stream state structure where the stream state is copied.

LeapfrogStream

Initializes a stream using the leapfrog method.

Syntax

Fortran:

```
call vslleapfrogstream( stream, k, nstreams )
```

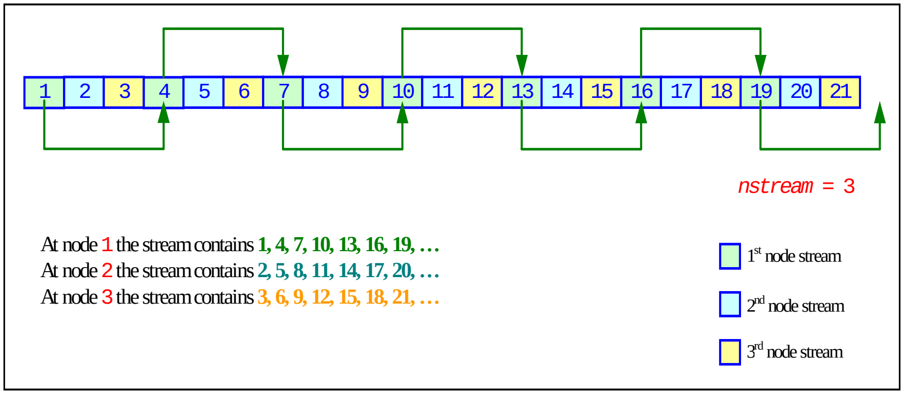
C:

```
vslLeapfrogStream( stream, k, nstreams )
```

Description

The function allows generating random numbers in a random stream with non-unit stride. This feature is particularly useful in distributing random numbers from original stream across *`nstreams`* buffers without generating the original random sequence with subsequent manual distribution. One of the important applications of the leapfrog method is slitting the original sequence into non-overlapping subsequences across *`nstreams`* computational nodes. The function initializes the original random stream (see Figure 10-1) to generate random numbers for the computational node k, $0 \le k < nstreams$, where `nstreams` is the largest number of computational nodes used.

Figure 10-1 Leapfrog Method



The leapfrog method is supported only for those basic generators that allow splitting elements by the leapfrog method, which is more efficient than simply generating them by a generator with subsequent manual distribution across computational nodes. See *VSL Notes* for details.

For quasi-random basic generators the leapfrog method allows generating individual components of quasi-random vectors instead of whole quasi-random vectors. In this case `nstreams` parameter should be equal to the dimension of the quasi-random vector while `k` parameter should be the index of a component to be generated ($0 \le k < nstreams$). Other parameters values are not allowed.

The following code examples illustrate the initialization of three independent streams using the leapfrog method:

Example 10-1 FORTRAN Code for Leapfrog Method

```
…
type(VSL_STREAM_STATE)stream1
type(VSL_STREAM_STATE)stream2
type(VSL_STREAM_STATE)stream3

! Creating 3 identical streams
call vslnewstream(stream1, VSL_BRNG_MCG31, 174)
call vslcopystream(stream2, stream1)
call vslcopystream(stream3, stream1)
```

Example 10-1 FORTRAN Code for Leapfrog Method (continued)

```
! Leapfrogging the streams
call vslleapfrogstream(stream1, 0, 3)
call vslleapfrogstream(stream2, 1, 3)
call vslleapfrogstream(stream3, 2, 3)

! Generating random numbers
…
! Deleting the streams
call vsldeletestream(stream1)
call vsldeletestream(stream2)
call vsldeletestream(stream3)
…
```

Example 10-2 C Code for Leapfrog Method

```
…
VSLStreamStatePtr stream1;
VSLStreamStatePtr stream2;
VSLStreamStatePtr stream3;

/* Creating 3 identical streams */
vslNewStream(&stream1, VSL_BRNG_MCG31, 174);
vslCopyStream(&stream2, stream1);
vslCopyStream(&stream3, stream1);

/* Leapfrogging the streams */
vslLeapfrogStream(stream1, 0, 3);
vslLeapfrogStream(stream2, 1, 3);
vslLeapfrogStream(stream3, 2, 3);

/* Generating random numbers */
…
/* Deleting the streams */
vslDeleteStream(&stream1);
vslDeleteStream(&stream2);
vslDeleteStream(&stream3);
…
```

Input Parameters

FORTRAN:

stream	`TYPE(VSL_STREAM_STATE), INTENT(IN)`. Descriptor of the stream to which the leapfrog method is applied.

k	`INTEGER, INTENT(IN)`. Index of the computational node, or stream number.
nstreams	`INTEGER, INTENT(IN)`. Largest number of computational nodes, or stride.

C:

stream	`VSLStreamStatePtr`. Pointer to the stream state structure to which the leapfrog method is applied.
k	`int`. Index of the computational node, or stream number.
nstreams	`int`. Largest number of computational nodes, or stride.

SkipAheadStream

Initializes a stream using the block-splitting method.

Syntax

Fortran:

```
call vslskipaheadstream( stream, nskip )
```

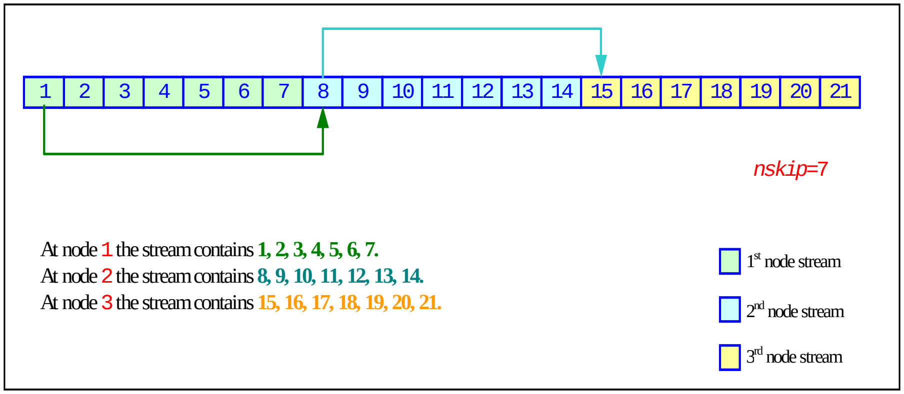
C:

```
vslSkipAheadStream( stream, nskip )
```

Description

This function skips a given number of elements in a random stream. This feature is particularly useful in distributing random numbers from original random stream across different computational nodes. If the largest number of random numbers used by a computational node is *`nskip`*, then the original random sequence may be split by `SkipAheadStream` into non-overlapping blocks of *nskip* size so that each block corresponds to the respective computational node. The number of computational nodes is unlimited. This method is known as the block-splitting method or as the skip-ahead method. (see Figure 10-2).

Figure 10-2 Block-Splitting Method



The skip-ahead method is supported only for those basic generators that allow skipping elements by the skip-ahead method, which is more efficient than simply generating them by generator with subsequent manual skipping. See *VSL Notes* for details.

Please note that for quasi-random basic generators the skip-ahead method works with components of quasi-random vectors rather than with whole quasi-random vectors. Thus to skip `NS` quasi-random vectors, set *nskip* parameter equal to the `NS*DIMEN`, where `DIMEN` is the dimension of quasi-random vector.

The following code examples illustrate how to initialize three independent streams using `SkipAheadStream` function:

Example 10-3 FORTRAN Code for Block-Splitting Method

```
…
TYPE(VSL_STREAM_STATE)stream1
TYPE(VSL_STREAM_STATE)stream2
TYPE(VSL_STREAM_STATE)stream3

! Creating the 1st stream
call vslnewstream(stream1, VSL_BRNG_MCG31, 174)

! Skipping ahead by 7 elements the 2nd stream
call vslcopystream(stream2, stream1);
```

Example 10-3 FORTRAN Code for Block-Splitting Method (continued)

```
call vslskipaheadstream(stream2, 7);

! Skipping ahead by 7 elements the 3rd stream
call vslcopystream(stream3, stream2);
call vslskipaheadstream(stream3, 7);

! Generating random numbers
…
! Deleting the streams
call vsldeletestream(stream1)
call vsldeletestream(stream2)
call vsldeletestream(stream3)
…
```

Example 10-4 C Code for Block-Splitting Method

```
VSLStreamStatePtr stream1;
VSLStreamStatePtr stream2;
VSLStreamStatePtr stream3;

/* Creating the 1st stream */
vslNewStream(&stream1, VSL_BRNG_MCG31, 174);

/* Skipping ahead by 7 elements the 2nd stream */
vslCopyStream(&stream2, stream1);
vslSkipAheadStream(stream2, 7);

/* Skipping ahead by 7 elements the 3rd stream */
vslCopyStream(&stream3, stream2);
vslSkipAheadStream(stream3, 7);

/* Generating random numbers */
…
/* Deleting the streams */
vslDeleteStream(&stream1);
vslDeleteStream(&stream2);
vslDeleteStream(&stream3);
…
```

Input Parameters

FORTRAN:

stream `TYPE(VSL_STREAM_STATE), INTENT(IN)`.
Descriptor of the stream to which the block-splitting method is applied.

nskip	`INTEGER, INTENT(IN)`. Number of skipped elements.

C:

stream	`VSLStreamStatePtr`. Pointer to the stream state structure to which the block-splitting method is applied.
nskip	`int`. Number of skipped elements.

GetStreamStateBrng

Returns index of a basic generator used for generation of a given random stream.

Syntax

Fortran:

```
brng = vslgetstreamstatebrng( stream )
```

C:

```
brng = vslGetStreamStateBrng( stream )
```

Description

This function retrieves the index of a basic generator used for generation of a given random stream.

Input Parameters

FORTRAN:

stream	`TYPE(VSL_STREAM_STATE), INTENT(IN)`. Descriptor of the stream state.

C:

stream	`VSLStreamStatePtr`. Pointer to the stream state structure.

Output Parameters

FORTRAN:

brng	`INTEGER`. Index of the basic generator assigned for the generation of *`stream`* ; negative in case of an error.

C:

brng	`int`. Index of the basic generator assigned for the generation of *`stream`* ; negative in case of an error.

GetNumRegBrng

Obtains the number of currently registered basic generators.

Syntax

Fortran:

`nregbrng` `= vslgetnumregbrngs( )`

C:

`nregbrng` `= vslGetNumRegBrngs(` *`void`* `)`

Description

This function obtains the number of currently registered basic generators. Whenever user registers a user-designed basic generator the number of registered basic generators is incremented. The maximum number of basic generators that can be registered is determined by `VSL_MAX_REG_BRNGS` parameter.

Output Parameters

FORTRAN:

nregbrngs	`INTEGER`. The number of basic generators registered at the moment of the function call.

C:

nregbrngs `int`. The number of basic generators registered at the moment of the function call.

Distribution Generators

This section contains description of VSL subroutines for generating random numbers with different types of distribution. Each function group is introduced by the type of underlying distribution and contains a short description of its functionality, as well as specifications of the call sequence for both FORTRAN and C-interface and the explanation of input and output parameters. Table 10-2 and Table 10-3 list the random number generator subroutines, together with used data types and output distributions.

Table 10-2 Continuous Distribution Generators

Type of Distribution	Data Types	Description
`Uniform`	`s, d`	Uniform continuous distribution on the interval [*a*,*b*].
`Gaussian`	`s, d`	Normal (Gaussian) distribution.
`GaussianMV`	`s, d`	Multivariate normal (Gaussian) distribution.
`Exponential`	`s, d`	Exponential distribution.
`Laplace`	`s, d`	Laplace distribution (double exponential distribution).
`Weibull`	`s, d`	Weibull distribution.
`Cauchy`	`s, d`	Cauchy distribution.
`Rayleigh`	`s, d`	Rayleigh distribution.
`Lognormal`	`s, d`	Lognormal distribution.
`Gumbel`	`s, d`	Gumbel (extreme value) distribution.

Table 10-3 Discrete Distribution Generators

Type of Distribution	Data Types	Description
`Uniform`	`i`	Uniform discrete distribution on the interval [*a*,*b*).
`UniformBits`	`i`	Generator of integer random values with uniform bit distribution.
`Bernoulli`	`i`	Bernoulli distribution.
`Geometric`	`i`	Geometric distribution.

Table 10-3 Discrete Distribution Generators (continued)

Type of Distribution	Data Types	Description
`Binomial`	i	Binomial distribution.
`Hypergeometric`	i	Hypergeometric distribution.
`Poisson`	i	Poisson distribution.
`PoissonV`	i	Poisson distribution with varying mean.
`NegBinomial`	i	Negative binomial distribution, or Pascal distribution.

Continuous Distributions

This section describes routines for generating random numbers with continuous distribution.

Uniform

Generates random numbers with uniform distribution.

Syntax

Fortran:

```
call vsrnguniform( method, stream, n, r, a, b )
call vdrnguniform( method, stream, n, r, a, b )
```

C:

```
vsRngUniform( method, stream, n, r, a, b )
vdRngUniform( method, stream, n, r, a, b )
```

Description

This function generates random numbers uniformly distributed over the interval [*a, b*], where *a, b* are the left and right bounds of the interval, respectively, and $a, b \in R$; $a < b$.

The probability density function is given by:

$$f_{a,b}(x) = \begin{cases} \dfrac{1}{b-a}, & x \in [a, b] \\ 0, & x \notin [a, b] \end{cases}, -\infty < x < +\infty .$$

The cumulative distribution function is as follows:

$$F_{a,b}(x) = \begin{cases} 0, & x < a \\ \dfrac{x-a}{b-a}, & a \le x < b \\ 1, & x \ge b \end{cases}, -\infty < x < +\infty .$$

Input Parameters

FORTRAN:

method	`INTEGER, INTENT(IN).` Generation method; dummy and set to 0 in case of uniform distribution.
stream	`TYPE (VSL_STREAM_STATE), INTENT(IN).` Descriptor of the stream state structure.
n	`INTEGER, INTENT(IN).` Number of random values to be generated.
a	`REAL, INTENT(IN)` for `vsrnguniform`. `DOUBLE PRECISION, INTENT(IN)` for `vdrnguniform`. Left bound *a*.
b	`REAL, INTENT(IN)` for `vsrnguniform`. `DOUBLE PRECISION, INTENT(IN)` for `vdrnguniform`. Right bound *b*.

C:

method	`int.` Generation method; dummy and set to 0 in case of uniform distribution.
stream	`VSLStreamStatePtr.` Pointer to the stream state structure.
n	`int.` Number of random values to be generated.

a `float` for `vsRngUniform`.

`double` for `vdRngUniform`.

Left bound *a*.

b `float` for `vsRngUniform`.

`double` for `vdRngUniform`.

Right bound *b*.

Output Parameters

FORTRAN:

r `REAL, INTENT(OUT)` for `vsrnguniform`.

`DOUBLE PRECISION, INTENT(OUT)` for `vdrnguniform`.

Vector of *n* random numbers uniformly distributed over the interval [*a*,*b*].

C:

r `float*` for `vsRngUniform`.

`double*` for `vdRngUniform`.

Vector of *n* random numbers uniformly distributed over the interval [*a*,*b*].

Gaussian

Generates normally distributed random numbers.

Syntax

Fortran:

```
call vsrnggaussian( method, stream, n, r, a, sigma )
call vdrnggaussian( method, stream, n, r, a, sigma )
```

C:

```
vsRngGaussian( method, stream, n, r, a, sigma )
vdRngGaussian( method, stream, n, r, a, sigma )
```

Description

This function generates random numbers with normal (Gaussian) distribution with mean value a and standard deviation σ, where

$$a, \sigma \in R ; \sigma > 0 .$$

The probability density function is given by:

$$f_{a,\sigma}(x) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{(x-a)^2}{2\sigma^2}\right), -\infty < x < +\infty .$$

The cumulative distribution function is as follows:

$$F_{a,\sigma}(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{(y-a)^2}{2\sigma^2}\right) dy, -\infty < x < +\infty .$$

The cumulative distribution function $F_{a,\sigma}(x)$ can be expressed in terms of standard normal distribution $\Phi(x)$ as

$$F_{a,\sigma}(x) = \Phi((x-a)/\sigma) .$$

Input Parameters

FORTRAN:

method	`INTEGER, INTENT(IN)`. Generation method.
stream	`TYPE (VSL_STREAM_STATE), INTENT(IN)`. Descriptor of the stream state structure.
n	`INTEGER, INTENT(IN)`. Number of random values to be generated.

a	`REAL, INTENT(IN)` for `vsrnggaussian`. `DOUBLE PRECISION, INTENT(IN)` for `vdrnggaussian`. Mean value *a*.
sigma	`REAL, INTENT(IN)` for `vsrnggaussian`. `DOUBLE PRECISION, INTENT(IN)` for `vdrnggaussian`. Standard deviation σ.

C:

method	`int`. Generation method.
stream	`VSLStreamStatePtr`. Pointer to the stream state structure.
n	`int`. Number of random values to be generated.
a	`float` for `vsRngGaussian`. `double` for `vdRngGaussian`. Mean value *a*.
sigma	`float` for `vsRngGaussian`. `double` for `vdRngGaussian`. Standard deviation σ.

Output Parameters

FORTRAN:

r	`REAL, INTENT(OUT)` for `vsrnggaussian`. `DOUBLE PRECISION, INTENT(OUT)` for `vdrnggaussian`. Vector of *n* normally distributed random numbers.

C:

r `float*` for `vsRngGaussian`.

`double*` for `vdRngGaussian`.

Vector of *n* normally distributed random numbers.

GaussianMV

Generates random numbers from multivariate normal distribution.

Syntax

Fortran:

```
call vsrnggaussianmv( method, stream, n, r, dimen, mstorage, a, t )
call vdrnggaussianmv( method, stream, n, r, dimen, mstorage, a, t )
```

C:

```
vsRngGaussianMV( method, stream, n, r, dimen, mstorage, a, T )
vdRngGaussianMV( method, stream, n, r, dimen, mstorage, a, T )
```

Description

This function generates random numbers with *d*-variate normal (Gaussian) distribution with mean value a and variance-covariance matrix *C*, where

$a \in R^d$; *C* is a $d \times d$ symmetric positive-definite matrix.

The probability density function is given by:

$$f_{a,C}(x) = \frac{1}{\sqrt{\det(2\pi C)}} \exp(-1/2(x-a)^T C^{-1}(x-a)), \text{ where } x \in R^d.$$

Matrix *C* can be represented as $C = TT^T$, where *T* is a lower triangular matrix - Cholesky factor of *C*.

Instead of variance-covariance matrix C the generation subroutines require Cholesky factor of C in input. To compute Cholesky factor of the matrix C, a user may call MKL LAPACK routines for matrix factorization: `?potrf` or `?pptrf` for `v?RngGaussianMV/v?rnggaussianmv` subroutines (`?` means either `s` or `d` for single and double precision respectively). See Application Notes below for more details.

Input Parameters

FORTRAN:

`method`	`INTEGER, INTENT(IN).` Generation method.
`stream`	`TYPE (VSL_STREAM_STATE), INTENT(IN).` Descriptor of the stream state structure.
`n`	`INTEGER, INTENT(IN).` Number of d-dimensional random vectors to be generated in a call.
`dimen`	`INTEGER, INTENT(IN).` Dimension d ($d \geq 1$) of output random vectors.
`mstorage`	`INTEGER, INTENT(IN).` Matrix storage scheme for upper triangular matrix T^T. Subroutine supports three matrix storage schemes: `VSL_MATRIX_STORAGE_FULL` – all $d \times d$ elements of the matrix T^T are passed, however, only the upper triangle part is actually used in the subroutine. `VSL_MATRIX_STORAGE_PACKED` – upper triangle elements of T^T are packed by rows into a one-dimensional array. `VSL_MATRIX_STORAGE_DIAGONAL` – only diagonal elements of T^T are passed.
`a`	`REAL, INTENT(IN)` for `vsrnggaussianmv`. `DOUBLE PRECISION, INTENT(IN)` for `vdrnggaussianvm`. Mean vector a of dimension d.

t `REAL, INTENT(IN)` for `vsrnggaussianmv`.

`DOUBLE PRECISION, INTENT(IN)` for `vdrnggaussianmv`.

Elements of the upper triangular matrix T^T passed according to the matrix storage scheme *mstorage*.

C:

method `int.` Generation method.

stream `VSLStreamStatePtr.` Pointer to the stream state structure.

n `int.` Number of random values to be generated.

dimen `int.` Dimension *d* ($d \geq 1$) of output random vectors.

mstorage `int.` Matrix storage scheme for lower triangular matrix T. Subroutine supports three matrix storage schemes:

- `VSL_MATRIX_STORAGE_FULL` – all $d \times d$ elements of the matrix T are passed, however, only the lower triangle part is actually used in the subroutine.
- `VSL_MATRIX_STORAGE_PACKED` – lower triangle elements of T are packed by columns into a one-dimensional array.
- `VSL_MATRIX_STORAGE_DIAGONAL` – only diagonal elements of T are passed.

a `float*` for `vsRngGaussianMV`.

`double*` for `vdRngGaussianMV`.

Mean vector a of dimension *d*.

sigma `float*` for `vsRngGaussianMV`.

`double*` for `vdRngGaussianMV`.

Elements of the lower triangular matrix T passed according to the matrix storage scheme *mstorage*.

Output Parameters

FORTRAN:

r	`REAL, INTENT(OUT)` for `vsrnggaussianmv`. `DOUBLE PRECISION, INTENT(OUT)` for `vdrnggaussianmv`. Array of *n* random vectors of dimension *dimen*.

C:

r	`float*` for `vsRngGaussianMV`. `double*` for `vdRngGaussianMV`. Array of *n* random vectors of dimension *dimen*.

Application Notes

Since matrices are stored in Fortran by columns, while in C they are stored by rows, the usage of MKL factorization subroutines (assuming Fortran matrices storage) in combination with multivariate normal RNG (assuming C matrix storage) is slightly different in C and Fortran. The following tables help in using these subroutines in C and Fortran. For further information please refer to the appropriate VSL example file.

Table 10-4 Using Cholesky Factorization Subroutines in Fortran

Matrix Storage Scheme	Variance-Covariance Matrix Argument	Factorization Subroutine	UPLO Parameter in Factorization Subroutine	Result of Factorization as Input Argument for RNG
`VSL_MATRIX_STORAGE_FULL`	*C* in Fortran two-dimentional array	`spotrf` for vsrnggaussianmv `dpotrf` for vdrnggaussianmv	'U'	Upper triangle of T^T. Lower triangle is not used.
`VSL_MATRIX_STORAGE_PACKED`	Lower triangle of *C* packed by columns into one-dimensional array	`spptrf` for vsrnggaussianmv `dpptrf` for vdrnggaussianmv	'L'	Upper triangle of T^T packed by rows into one-dimentional array.

Table 10-5 Using Cholesky Factorization Subroutines in C

Matrix Storage Scheme	Variance-Covariance Matrix Argument	Factorization Subroutine	UPLO Parameter in Factorization Subroutine	Result of Factorization as Input Argument for RNG
`VSL_MATRIX_STORAGE_FULL`	*C* in C two-dimentional array	`spotrf` for vsRngGaussianMV `dpotrf` for vdRngGaussianMV	'U'	Upper triangle of T^T. Lower triangle is not used.
`VSL_MATRIX_STORAGE_PACKED`	Lower triangle of *C* packed by columns into one-dimensional array	`spptrf` for vsRngGaussianMV `dpptrf` for vdRngGaussianMV	'L'	Upper triangle of T^T packed by rows into one-dimentional array.

Exponential

Generates exponentially distributed random numbers.

Syntax

Fortran:

```
call vsrngexponential( method, stream, n, r, a, beta )
call vdrngexponential( method, stream, n, r, a, beta )
```

C:

```
vsRngExponential( method, stream, n, r, a, beta )
vdRngExponential( method, stream, n, r, a, beta )
```

Description

This function generates random numbers with exponential distribution that has the displacement a and scalefactor β, where $a, \beta \in R$; $\beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{1}{\beta}\exp((-(x-a))/\beta), & x \ge a \\ 0, & x < a \end{cases}, -\infty < x < +\infty .$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp((-(x-a))/\beta), & x \ge a \\ 0, & x < a \end{cases}, -\infty < x < +\infty .$$

Input Parameters

FORTRAN:

method — `INTEGER, INTENT(IN).` Generation method.

stream — `TYPE (VSL_STREAM_STATE), INTENT(IN).` Descriptor of the stream state structure.

n — `INTEGER, INTENT(IN).` Number of random values to be generated.

a — `REAL, INTENT(IN)` for `vsrngexponential`.

`DOUBLE PRECISION, INTENT(IN)` for `vdrngexponential`.

Displacement *a*.

beta — `REAL, INTENT(IN)` for `vsrngexponential`.

`DOUBLE PRECISION, INTENT(IN)` for `vdrngexponential`.

Scalefactor β.

C:

method — `int.` Generation method.

stream — `VSLStreamStatePtr.` Pointer to the stream state structure.

n `int.` Number of random values to be generated.

a `float` for `vsRngExponential`.

`double` for `vdRngExponential`.

Displacement *a*.

beta `float` for `vsRngExponential`.

`double` for `vdRngExponential`.

Scalefactor β.

Output Parameters

FORTRAN:

r `REAL, INTENT(OUT)` for `vsrngexponential`.

`DOUBLE PRECISION, INTENT(OUT)` for `vdrngexponential`.

Vector of *n* exponentially distributed random numbers.

C:

r `float*` for `vsRngExponential`.

`double*` for `vdRngExponential`.

Vector of *n* exponentially distributed random numbers.

Laplace

Generates random numbers with Laplace distribution.

Syntax

Fortran:

```
call vsrnglaplace( method, stream, n, r, a, beta )
call vdrnglaplace( method, stream, n, r, a, beta )
```

C:

```
vsRngLaplace( method, stream, n, r, a, beta )
vdRngLaplace( method, stream, n, r, a, beta )
```

Description

This function generates random numbers with Laplace distribution with mean value (or average) a and scalefactor β, where

$a, \beta \in R$; $\beta > 0$. The scalefactor value determines the standard deviation as

$\sigma = \beta\sqrt{2}$.

The probability density function is given by:

$$f_{a,\beta}(x) = \frac{1}{\sqrt{2}\beta}\exp\left(-\frac{|x-a|}{\beta}\right), -\infty < x < +\infty .$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} \frac{1}{2}\exp\left(-\frac{|x-a|}{\beta}\right), & x < a \\ 1 - \frac{1}{2}\exp\left(-\frac{|x-a|}{\beta}\right), & x \geq a \end{cases}, -\infty < x < +\infty .$$

Input Parameters

FORTRAN:

method	`INTEGER, INTENT(IN)`. Generation method.
stream	`TYPE (VSL_STREAM_STATE), INTENT(IN)`. Descriptor of the stream state structure.
n	`INTEGER, INTENT(IN)`. Number of random values to be generated.
a	`REAL, INTENT(IN)` for `vsrnglaplace`. `DOUBLE PRECISION, INTENT(IN)` for `vdrnglaplace`. Mean value *a*.
beta	`REAL, INTENT(IN)` for `vsrnglaplace`. `DOUBLE PRECISION, INTENT(IN)` for `vdrnglaplace`. Scalefactor β.

C:

method	`int`. Generation method.
stream	`VSLStreamStatePtr`. Pointer to the stream state descriptor.
n	`int`. Number of random values to be generated.
a	`float` for `vsRngLaplace`. `double` for `vdRngLaplace`. Mean value *a*.
beta	`float` for `vsRngLaplace`. `double` for `vdRngLaplace`. Scalefactor β.

Output Parameters

FORTRAN:

r — `REAL, INTENT(OUT)` for `vsrnglaplace`.

`DOUBLE PRECISION, INTENT(OUT)` for `vdrnglaplace`.

Vector of *n* Laplace distributed random numbers.

C:

r — `float*` for `vsRngLaplace`.

`double*` for `vdRngLaplace`.

Vector of *n* Laplace distributed random numbers.

Weibull

Generates Weibull distributed random numbers.

Syntax

Fortran:

```
call vsrngweibull( method, stream, n, r, alpha, a, beta )
call vdrngweibull( method, stream, n, r, alpha, a, beta )
```

C:

```
vsRngWeibull( method, stream, n, r, alpha, a, beta )
vdRngWeibull( method, stream, n, r, alpha, a, beta )
```

Description

This function generates Weibull distributed random numbers with displacement a, scalefactor β, and shape α, where $\alpha, \beta, a \in R$; $\alpha > 0$; $\beta > 0$.

The probability density function is given by:

$$f_{a,\alpha,\beta}(x) = \begin{cases} \dfrac{\alpha}{\beta^{\alpha}}(x-a)^{\alpha-1}\exp\left(-\left(\dfrac{x-a}{\beta}\right)^{\alpha}\right), & x \ge a \\ 0, & x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\alpha,\beta}(x) = \begin{cases} 1-\exp\left(-\left(\dfrac{x-a}{\beta}\right)^{\alpha}\right), & x \ge a \\ 0, & x < a \end{cases}, -\infty < x < +\infty .$$

Input Parameters

FORTRAN:

method	`INTEGER, INTENT(IN)`. Generation method.
stream	`TYPE (VSL_STREAM_STATE), INTENT(IN)`. Descriptor of the stream state structure.
n	`INTEGER, INTENT(IN)`. Number of random values to be generated.
alpha	`REAL, INTENT(IN)` for `vsrngweibull`. `DOUBLE PRECISION, INTENT(IN)` for `vdrngweibull`. Shape α.
a	`REAL, INTENT(IN)` for `vsrngweibull`. `DOUBLE PRECISION, INTENT(IN)` for `vdrngweibull`. Displacement *a*.

beta	`REAL, INTENT(IN)` for `vsrngweibull`. `DOUBLE PRECISION, INTENT(IN)` for `vdrngweibull`. Scalefactor β.

C:

method	`int.` Generation method.
stream	`VSLStreamStatePtr.` Pointer to the stream state structure.
n	`int.` Number of random values to be generated.
alpha	`float` for `vsRngWeibull`. `double` for `vdRngWeibull`. Shape α.
a	`float` for `vsRngWeibull`. `double` for `vdRngWeibull`. Displacement *a*.
beta	`float` for `vsRngWeibull`. `double` for `vdRngWeibull`. Scalefactor β.

Output Parameters

FORTRAN:

r	`REAL, INTENT(OUT)` for `vsrngweibull`. `DOUBLE PRECISION, INTENT(OUT)` for `vdrngweibull`. Vector of *n* Weibull distributed random numbers.

C:

r `float*` for `vsRngWeibull`.

`double*` for `vdRngWeibull`.

Vector of *n* Weibull distributed random numbers.

Cauchy

Generates Cauchy distributed random values.

Syntax

Fortran:

```
call vsrngcauchy( method, stream, n, r, a, beta )
call vdrngcauchy( method, stream, n, r, a, beta )
```

C:

```
vsRngCauchy( method, stream, n, r, a, beta )
vdRngCauchy( method, stream, n, r, a, beta )
```

Description

This function generates Cauchy distributed random numbers with displacement a and scalefactor β, where $a, \beta \in R$; $\beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \frac{1}{\pi\beta\left(1 + \left(\frac{x-a}{\beta}\right)^2\right)}, -\infty < x < +\infty .$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \frac{1}{2} + \frac{1}{\pi}\arctan\left(\frac{x-a}{\beta}\right), -\infty < x < +\infty .$$

Input Parameters

FORTRAN:

method	`INTEGER, INTENT(IN).` Generation method.
stream	`TYPE (VSL_STREAM_STATE), INTENT(IN).` Descriptor of the stream state structure.
n	`INTEGER, INTENT(IN).` Number of random values to be generated.
a	`REAL, INTENT(IN)` for `vsrngcauchy`. `DOUBLE PRECISION, INTENT(IN)` for `vdrngcauchy`. Displacement *a*.
beta	`REAL, INTENT(IN)` for `vsrngcauchy`. `DOUBLE PRECISION, INTENT(IN)` for `vdrngcauchy`. Scalefactor β.

C:

method	`int.` Generation method.
stream	`VSLStreamStatePtr.` Pointer to the stream state structure.
n	`int.` Number of random values to be generated.
a	`float` for `vsRngCauchy`. `double` for `vdRngCauchy`. Displacement *a*.
beta	`float` for `vsRngCauchy`. `double` for `vdRngCauchy`. Scalefactor β.

Output Parameters

FORTRAN:

r — `REAL, INTENT(OUT)` for `vsrngcauchy`.

`DOUBLE PRECISION, INTENT(OUT)` for `vdrngcauchy`.

Vector of *n* Cauchy distributed random numbers.

C:

r — `float*` for `vsRngCauchy`.

`double*` for `vdRngCauchy`.

Vector of *n* Cauchy distributed random numbers.

Rayleigh

Generates Rayleigh distributed random values.

Syntax

Fortran:

```
call vsrngrayleigh( method, stream, n, r, a, beta )
call vdrngrayleigh( method, stream, n, r, a, beta )
```

C:

```
vsRngRayleigh( method, stream, n, r, a, beta )
vdRngRayleigh( method, stream, n, r, a, beta )
```

Description

This function generates Rayleigh distributed random numbers with displacement a and scalefactor β, where $a, \beta \in R$; $\beta > 0$.

Rayleigh distribution is a special case of Weibull distribution, where the shape parameter $\alpha = 2$.

The probability density function is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{2(x-a)}{\beta^2}\exp\left(-\frac{(x-a)^2}{\beta^2}\right), & x \ge a \\ 0, & x < a \end{cases}, -\infty < x < +\infty .$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1-\exp\left(-\frac{(x-a)^2}{\beta^2}\right), & x \ge a \\ 0, & x < a \end{cases}, -\infty < x < +\infty .$$

Input Parameters

FORTRAN:

method	`INTEGER, INTENT(IN).` Generation method.
stream	`TYPE (VSL_STREAM_STATE), INTENT(IN).` Descriptor of the stream state structure.
n	`INTEGER, INTENT(IN).` Number of random values to be generated.
a	`REAL, INTENT(IN)` for `vsrngrayleigh`. `DOUBLE PRECISION, INTENT(IN)` for `vdrngrayleigh`. Displacement a.
beta	`REAL, INTENT(IN)` for `vsrngrayleigh`. `DOUBLE PRECISION, INTENT(IN)` for `vdrngrayleigh`. Scalefactor β.

C:

method	`int.` Generation method.

stream	`VSLStreamStatePtr.` Pointer to the stream state structure.
n	`int.` Number of random values to be generated.
a	`float` for `vsRngRayleigh`. `double` for `vdRngRayleigh`. Displacement *a*.
beta	`float` for `vsRngRayleigh`. `double` for `vdRngRayleigh`. Scalefactor β.

Output Parameters

FORTRAN:

r	`REAL, INTENT(OUT)` for `vsrngrayleigh`. `DOUBLE PRECISION, INTENT(OUT)` for `vdrngrayleigh`. Vector of *n* Rayleigh distributed random numbers.

C:

r	`float*` for `vsRngRayleigh`. `double*` for `vdRngRayleigh`. Vector of *n* Rayleigh distributed random numbers.

Lognormal

Generates lognormally distributed random numbers.

Syntax

Fortran:

```
call vsrnglognormal( method, stream, n, r, a, sigma, b, beta )
call vdrnglognormal( method, stream, n, r, a, sigma, b, beta )
```

C:

```
vsRngLognormal( method, stream, n, r, a, sigma, b, beta )
vdRngLognormal( method, stream, n, r, a, sigma, b, beta )
```

Discussion

This function generates lognormally distributed random numbers with average of distribution a and standard deviation σ of subject normal distribution, displacement b, and scalefactor β, where

$$a, \sigma, b, \beta \in R \; ; \sigma > 0 ; \beta > 0 .$$

The probability density function is given by:

$$f_{a, \sigma, b, \beta}(x) = \begin{cases} \dfrac{1}{\sigma(x-b)\sqrt{2\pi}} \exp\left(-\dfrac{[\ln((x-b)/\beta)-a]^2}{2\sigma^2}\right), & x > b \\ 0, & x \le b \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a, \sigma, b, \beta}(x) = \begin{cases} \Phi((\ln((x-b)/\beta)-a)/\sigma), & x > b \\ 0, & x \le b \end{cases}$$

Input Parameters

FORTRAN:

method	`INTEGER, INTENT(IN)`. Generation method.
stream	`TYPE (VSL_STREAM_STATE), INTENT(IN)`. Descriptor of the stream state structure.
n	`INTEGER, INTENT(IN)`. Number of random values to be generated.
a	`REAL, INTENT(IN)` for `vsrnglognormal`. `DOUBLE PRECISION, INTENT(IN)` for `vdrnglognormal`. Average *a* of the subject normal distribution.
sigma	`REAL, INTENT(IN)` for `vsrnglognormal`. `DOUBLE PRECISION, INTENT(IN)` for `vdrnglognormal`. Standard deviation σ of the subject normal distribution.
b	`REAL, INTENT(IN)` for `vsrnglognormal`. `DOUBLE PRECISION, INTENT(IN)` for `vdrnglognormal`. Displacement *b*.
beta	`REAL, INTENT(IN)` for `vsrnglognormal`. `DOUBLE PRECISION, INTENT(IN)` for `vdrnglognormal`. Scalefactor value β.

C:

method	`int`. Generation method.
stream	`VSLStreamStatePtr`. Pointer to the stream state structure.
n	`int`. Number of random values to be generated.

a	`float` for `vsRngLognormal`. `double` for `vdRngLognormal`. Average *a* of the subject normal distribution.
sigma	`float` for `vsRngLognormal`. `double` for `vdRngLognormal`. Standard deviation σ of the subject normal distribution.
b	`float` for `vsRngLognormal`. `double` for `vdRngLognormal`. Displacement *b*.
beta	`float` for `vsRngLognormal`. `double` for `vdRngLognormal`. Scalefactor value β.

Output Parameters

FORTRAN:

r	`REAL, INTENT(OUT)` for `vsrnglognormal`. `DOUBLE PRECISION, INTENT(OUT)` for `vdrnglognormal`. Vector of *n* lognormally distributed random numbers.

C:

r	`float*` for `vsRngLognormal`. `double*` for `vdRngLognormal`. Vector of *n* lognormally distributed random numbers.

Gumbel

Generates Gumbel distributed random values.

Syntax

Fortran:

```
call vsrnggumbel( method, stream, n, r, a, beta )
call vdrnggumbel( method, stream, n, r, a, beta )
```

C:

```
vsRngGumbel( method, stream, n, r, a, beta )
vdRngGumbel( method, stream, n, r, a, beta )
```

Description

This function generates Gumbel distributed random numbers with displacement a and scalefactor β, where $a, \beta \in R$; $\beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \frac{1}{\beta}\exp\left(\frac{x-a}{\beta}\right)\exp(-\exp((x-a)/\beta)), -\infty < x < +\infty .$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = 1 - \exp(-\exp((x-a)/\beta)), -\infty < x < +\infty .$$

Input Parameters

FORTRAN:

method	`INTEGER, INTENT(IN)`. Generation method.
stream	`TYPE (VSL_STREAM_STATE), INTENT(IN)`. Descriptor of the stream state structure.
n	`INTEGER, INTENT(IN)`. Number of random values to be generated.

a	`REAL, INTENT(IN)` for `vsrnggumbel`. `DOUBLE PRECISION, INTENT(IN)` for `vdrnggumbel`. Displacement *a*.
beta	`REAL, INTENT(IN)` for `vsrnggumbel`. `DOUBLE PRECISION, INTENT(IN)` for `vdrnggumbel`. Scalefactor β.

C:

method	`int`. Generation method.
stream	`VSLStreamStatePtr`. Pointer to the stream state structure.
n	`int`. Number of random values to be generated.
a	`float` for `vsRngGumbel`. `double` for `vdRngGumbel`. Displacement *a*.
beta	`float` for `vsRngGumbel`. `double` for `vdRngGumbel`. Scalefactor β.

Output Parameters

FORTRAN:

r	`REAL, INTENT(OUT)` for `vsrnggumbel`. `DOUBLE PRECISION, INTENT(OUT)` for `vdrnggumbel`. Vector of *n* random values with Gumbel distribution.

C:

r `float*` for `vsRngGumbel`.

`double*` for `vdRngGumbel`.

Vector of *n* random values with Gumbel distribution.

Discrete Distributions

This section describes routines for generating random numbers with discrete distribution.

Uniform

Generates random numbers uniformly distributed over the interval $[a, b)$.

Syntax

Fortran:

```
call virnguniform( method, stream, n, r, a, b )
```

C:

```
viRngUniform( method, stream, n, r, a, b )
```

Description

This function generates random numbers uniformly distributed over the interval $[a, b)$, where *a, b* are the left and right bounds of the interval, respectively, and $a, b \in Z$; $a < b$.

The probability distribution is given by:

$$P(X = k) = \frac{1}{b-a}, k \in \{a, a+1, \ldots, b-1\}.$$

The cumulative distribution function is as follows:

$$F_{a,b}(x) = \begin{cases} 0, & x < a \\ \dfrac{\lfloor x - a + 1 \rfloor}{b - a}, & a \le x < b, x \in R. \\ 1, & x \ge b \end{cases}$$

Input Parameters

FORTRAN:

method	`INTEGER, INTENT(IN).` Generation method.
stream	`TYPE (VSL_STREAM_STATE), INTENT(IN).` Descriptor of the stream state structure.
n	`INTEGER, INTENT(IN).` Number of random values to be generated.
a	`INTEGER, INTENT(IN).` Left interval bound *a*.
b	`INTEGER, INTENT(IN).` Right interval bound *b*.

C:

method	`int.` Generation method.
stream	`VSLStreamStatePtr.` Pointer to the stream state structure.
n	`int.` Number of random values to be generated.
a	`int.` Left interval bound *a*.
b	`int.` Right interval bound *b*.

Output Parameters

FORTRAN:

r	`INTEGER, INTENT(OUT).` Vector of *n* random values uniformly distributed over the interval $[a,b)$.

C:

r	`int*.` Vector of *n* random values uniformly distributed over the interval $[a,b)$.

UniformBits

Generates integer random values with uniform bit distribution.

Syntax

Fortran:

```
call virnguniformbits( method, stream, n, r )
```

C:

```
viRngUniformBits( method, stream, n, r )
```

Description

This function generates integer random values with uniform bit distribution.The generators of uniformly distributed numbers can be represented as recurrence relations over integer values in modular arithmetic. Apparently, each integer can be treated as a vector of several bits. In a truly random generator, these bits are random, while in pseudorandom generators this randomness can be violated. For example, a well known drawback of linear congruential generators is that lower bits are less random than higher bits (for example, see [Knuth81]). For this reason, care should be taken when using this function. Typically, in a 32-bit *LCG* only 24 higher bits of an integer value can be considered random. See *VSL Notes* for details.

Input Parameters

FORTRAN:

method	`INTEGER, INTENT(IN)`. Generation method. A dummy argument in `virnguniformbits`. Should be zero.
stream	`TYPE (VSL_STREAM_STATE), INTENT(IN)`. Descriptor of the stream state structure.
n	`INTEGER, INTENT(IN)`. Number of random values to be generated.

C:

method	`int`. Generation method. A dummy argument in `viRngUniformBits`. Should be zero.

stream	`VSLStreamStatePtr`. Pointer to the stream state structure.
n	`int`. Number of random values to be generated.

Output Parameters

FORTRAN:

r	`INTEGER, INTENT(OUT)`. Vector of *n* random integer numbers. If the *stream* was generated by a 64 or a 128-bit generator, each integer value is represented by two or four elements of *r* respectively. The number of bytes occupied by each integer is contained in the field *wordsize* of the structure `VSL_BRNG_PROPERTIES`. The total number of bits that are actually used to store the value are contained in the field *nbits* of the same structure. See "Advanced Service Subroutines" for a more detailed discussion of `VSL_BRNG_PROPERTIES`.

C:

r	`unsigned int*`. Vector of *n* random integer numbers. If the *stream* was generated by a 64 or a 128-bit generator, each integer value is represented by two or four elements of *r* respectively. The number of bytes occupied by each integer is contained in the field *WordSize* of the structure `VSLBrngProperties`. The total number of bits that are actually used to store the value are contained in the field *NBits* of the same structure. See "Advanced Service Subroutines" for a more detailed discussion of `VSLBrngProperties`.

Bernoulli

Generates Bernoulli distributed random values.

Syntax

Fortran:

```
call virngbernoulli( method, stream, n, r, p )
```

C:

```
viRngBernoulli( method, stream, n, r, p )
```

Description

This function generates Bernoulli distributed random numbers with probability p of a single trial success, where

$$p \in R; 0 \le p \le 1.$$

A variate is called Bernoulli distributed, if after a trial it is equal to 1 with probability of success *p*, and to 0 with probability `1-p`.

The probability distribution is given by:

$$P(X = 1) = p,$$

$$P(X = 0) = 1 - p.$$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, & x < 0 \\ 1 - p, & 0 \le x < 1 \\ 1, & x \ge 1 \end{cases}, x \in R.$$

Input Parameters

FORTRAN:

method	`INTEGER, INTENT(IN)`. Generation method.
stream	`TYPE (VSL_STREAM_STATE), INTENT(IN)`. Descriptor of the stream state structure.

n	`INTEGER, INTENT(IN).` Number of random values to be generated.
p	`DOUBLE PRECISION, INTENT(IN).` Success probability *p* of a trial.

C:

method	`int.` Generation method.
stream	`VSLStreamStatePtr.` Pointer to the stream state structure.
n	`int.` Number of random values to be generated.
p	`double.` Success probability *p* of a trial.

Output Parameters

FORTRAN:

r	`INTEGER, INTENT(OUT).` Vector of *n* Bernoulli distributed random values.

C:

r	`int*.` Vector of *n* Bernoulli distributed random values.

Geometric

Generates geometrically distributed random values.

Syntax

Fortran:

```
call virnggeometric( method, stream, n, r, p )
```

C:

```
viRngGeometric( method, stream, n, r, p )
```

Description

This function generates geometrically distributed random numbers with probability p of a single trial success, where $p \in R$; $0 < p < 1$.

A geometrically distributed variate represents the number of independent Bernoulli trials preceding the first success. The probability of a single Bernoulli trial success is p.

The probability distribution is given by:

$$P(X = k) = p \cdot (1 - p)^k, \; k \in \{0, 1, 2, \ldots\}.$$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, & x < 0 \\ 1 - (1 - p)^{\lfloor x + 1 \rfloor}, & x \ge 0 \end{cases}, \; x \in R.$$

Input Parameters

FORTRAN:

method	`INTEGER, INTENT(IN)`. Generation method.
stream	`TYPE (VSL_STREAM_STATE), INTENT(IN)`. Descriptor of the stream state structure.
n	`INTEGER, INTENT(IN)`. Number of random values to be generated.
p	`DOUBLE PRECISION, INTENT(IN)`. Success probability p of a trial.

C:

method	`int`. Generation method.
stream	`VSLStreamStatePtr`. Pointer to the stream state structure.
n	`int`. Number of random values to be generated.
p	`double`. Success probability p of a trial.

Output Parameters

FORTRAN:

r	`INTEGER, INTENT(OUT)`. Vector of *n* geometrically distributed random values.

C:

r	`int*`. Vector of *n* geometrically distributed random values.

Binomial

Generates binomially distributed random numbers.

Syntax

Fortran:

```
call virngbinomial( method, stream, n, r, ntrial, p )
```

C:

```
viRngBinomial( method, stream, n, r, ntrial, p )
```

Discussion

This function generates binomially distributed random numbers with number of independent Bernoulli trials m, and with probability p of a single trial success, where $p \in R$; $0 \le p \le 1$, $m \in N$.

A binomially distributed variate represents the number of successes in m independent Bernoulli trials with probability of a single trial success p.

The probability distribution is given by:

$$P(X = k) = C_m^k p^k (1-p)^{m-k}, k \in \{0, 1, \dots, m\}.$$

The cumulative distribution function is as follows:

$$F_{m,p}(x) = \begin{cases} 0, & x < 0 \\ \sum_{k=0}^{\lfloor x \rfloor} C_m^k p^k (1-p)^{m-k}, & 0 \le x < m \\ 1, & x \ge m \end{cases}, \quad x \in R.$$

Input Parameters

FORTRAN:

method	`INTEGER, INTENT(IN)`. Generation method.
stream	`TYPE (VSL_STREAM_STATE), INTENT(IN)`. Descriptor of the stream state structure.
n	`INTEGER, INTENT(IN)`. Number of random values to be generated.
ntrial	`INTEGER, INTENT(IN)`. Number of independent trials *m*.
p	`DOUBLE PRECISION, INTENT(IN)`. Success probability *p* of a single trial.

C:

method	`int`. Generation method.
stream	`VSLStreamStatePtr`. Pointer to the stream state structure.
n	`int`. Number of random values to be generated.
ntrial	`int`. Number of independent trials *m*.
p	`double`. Success probability *p* of a single trial.

Output Parameters

FORTRAN:

r	`INTEGER, INTENT(OUT)`. Vector of *n* binomially distributed random values.

C:

r	`int*`. Vector of n binomially distributed random values.

Hypergeometric

Generates hypergeometrically distributed random values.

Syntax

Fortran:

```
call virnghypergeometric( method, stream, n, r, l, s, m )
```

C:

```
viRngHypergeometric( method, stream, n, r, l, s, m )
```

Description

This function generates hypergeometrically distributed random values with lot size l, size of sampling s, and number of marked elements in the lot m, where $l, m, s \in N \cup \{0\}$; $l \geq \max(s, m)$.

Consider a lot of l elements comprising m "marked" and l-m "unmarked" elements. A trial sampling without replacement of exactly s elements from this lot helps to define the hypergeometric distribution, which is the probability that the group of s elements contains exactly k marked elements.

The probability distribution is given by:

$$P(X = k) = \frac{C_m^k C_{l-m}^{s-k}}{C_l^s}, \quad k \in \{\max(0, s+m-l), \ldots, \min(s, m)\}.$$

The cumulative distribution function is as follows:

$$F_{l,s,m}(x) = \begin{cases} 0, & x < \max(0, s+m-l) \\ \displaystyle\sum_{k=\max(0,s+m-l)}^{\lfloor x \rfloor} \frac{C_m^k C_{l-m}^{s-k}}{C_l^s}, & \max(0, s+m-l) \le x \le \min(s, m) \\ 1, & x > \min(s, m) \end{cases}$$

Input Parameters

FORTRAN:

method	`INTEGER, INTENT(IN).` Generation method.
stream	`TYPE (VSL_STREAM_STATE), INTENT(IN).` Descriptor of the stream state structure.
n	`INTEGER, INTENT(IN).` Number of random values to be generated.
l	`INTEGER, INTENT(IN).` Lot size *l*.
s	`INTEGER, INTENT(IN).` Size of sampling without replacement *s*.
m	`INTEGER, INTENT(IN).` Number of marked elements *m*.

C:

method	`int.` Generation method.
stream	`VSLStreamStatePtr.` Pointer to the stream state structure.
n	`int.` Number of random values to be generated.
l	`int.` Lot size *l*.
s	`int.` Size of sampling without replacement *s*.
m	`int.` Number of marked elements *m*.

Output Parameters

FORTRAN:

r	`INTEGER, INTENT(OUT)`. Vector of *n* hypergeometrically distributed random values.

C:

r	`int*`. Vector of *n* hypergeometrically distributed random values.

Poisson

Generates Poisson distributed random values.

Syntax

Fortran:

```
call virngpoisson( method, stream, n, r, lambda )
```

C:

```
viRngPoisson( method, stream, n, r, lambda )
```

Description

This function generates Poisson distributed random numbers with distribution parameter λ, where $\lambda \in R; \lambda > 0$.

The probability distribution is given by:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}, k \in \{0, 1, 2, \ldots\}.$$

The cumulative distribution function is as follows:

$$F_{\lambda}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda^k e^{-\lambda}}{k!}, & x \ge 0 \\ 0, & x < 0 \end{cases}, \quad x \in R.$$

Input Parameters

FORTRAN:

method	`INTEGER, INTENT(IN)`. Generation method.
stream	`TYPE (VSL_STREAM_STATE), INTENT(IN)`. Descriptor of the stream state structure.
n	`INTEGER, INTENT(IN)`. Number of random values to be generated.
lambda	`DOUBLE PRECISION, INTENT(IN)`. Distribution parameter λ.

C:

method	`int`. Generation method.
stream	`VSLStreamStatePtr`. Pointer to the stream state structure.
n	`int`. Number of random values to be generated.
lambda	`double`. Distribution parameter λ.

Output Parameters

FORTRAN:

r	`INTEGER, INTENT(OUT)`. Vector of *n* Poisson distributed random values.

C:

r	`int*`. Vector of *n* Poisson distributed values.

PoissonV

Generates Poisson distributed random values with varying mean.

Syntax

Fortran:

```
call virngpoissonv( method, stream, n, r, lambda )
```

C:

```
viRngPoissonV( method, stream, n, r, lambda )
```

Description

This function generates n Poisson distributed random numbers $X_i (i = 1, \ldots, n)$ with distribution parameter λ_i, where $\lambda_i \in R$; $\lambda_i > 0$.

The probability distribution is given by:

$$P(X_i = k) = \frac{\lambda_i^k \exp(-\lambda_i)}{k!}, k \in \{0, 1, 2, \ldots \}.$$

The cumulative distribution function is as follows:

$$F_{\lambda_i}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda_i^k e^{-\lambda_i}}{k!}, & x \ge 0 \\ 0, & x < 0 \end{cases}, x \in R.$$

Input Parameters

FORTRAN:

method	`INTEGER, INTENT(IN).` Generation method.
stream	`TYPE (VSL_STREAM_STATE), INTENT(IN).` Descriptor of the stream state structure.

n	`INTEGER, INTENT(IN).` Number of random values to be generated.
lambda	`DOUBLE PRECISION, INTENT(IN)`. Array of *n* distribution parameters λ_i.

C:

method	`int.` Generation method.
stream	`VSLStreamStatePtr.` Pointer to the stream state structure.
n	`int.` Number of random values to be generated.
lambda	`double*.` Array of *n* distribution parameters λ_i.

Output Parameters

FORTRAN:

r	`INTEGER, INTENT(OUT)`. Vector of *n* Poisson distributed random values.

C:

r	`int*.` Vector of *n* Poisson distributed random values.

NegBinomial

Generates random numbers with negative binomial distribution.

Syntax

Fortran:

```
call virngnegbinomial( method, stream, n, r, a, p )
```

C:

```
viRngNegBinomial( method, stream, n, r, a, p )
```

Description

This function generates random numbers with negative binomial distribution and distribution parameters a and p., where $p, a \in R$; $0 < p < 1$; $a > 0$.

If the first distribution parameter $a \in N$, this distribution is the same as Pascal distribution. If $a \in N$, the distribution can be interpreted as the expected time of a-th success in a sequence of Bernoulli trials, when the probability of success is p.

The probability distribution is given by:

$$P(X = k) = C_{a+k-1}^{k} p^{a}(1-p)^{k}, k \in \{0, 1, 2, \ldots\}.$$

The cumulative distribution function is as follows:

$$F_{a,p}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} C_{a+k-1}^{k} p^{a}(1-p)^{k}, & x \ge 0 \\ 0, & x < 0 \end{cases}, x \in R.$$

Input Parameters

FORTRAN:

method	`INTEGER, INTENT(IN).` Generation method.
stream	`TYPE (VSL_STREAM_STATE), INTENT(IN).` Descriptor of the stream state structure.
n	`INTEGER, INTENT(IN).` Number of random values to be generated.
a	`DOUBLE PRECISION, INTENT(IN)`. The first distribution parameter *a*.
p	`DOUBLE PRECISION, INTENT(IN)`. The second distribution parameter *p*.

C:

method	`int.` Generation method.
stream	`VSLStreamStatePtr.` Pointer to the stream state structure.

n	`int.` Number of random values to be generated.
a	`double`. The first distribution parameter *a*.
p	`double`. The second distribution parameter *p*.

Output Parameters

FORTRAN:

r	`INTEGER, INTENT(OUT)`. Vector of *n* random values with negative binomial distribution.

C:

r	`int*.` Vector of *n* random values with negative binomial distribution.

Advanced Service Subroutines

This section describes service subroutines for registering a user-designed basic generator (`RegisterBrng`) and for obtaining properties of the previously registered basic generators (`GetBrngProperties`). See *VSL Notes* ("Basic Generators" section of VSL Structure chapter) for substantiation of the need for several basic generators including user-defined BRNGs.

Data types

The subroutines of this section refer to a structure defining the properties of the basic generator. This structure is described in Fortran as follows:

```
TYPE VSL_BRNG_PROPERTIES
    INTEGER streamstatesize
    INTEGER nseeds
    INTEGER includeszero
    INTEGER wordsize
    INTEGER nbits
    INTEGER initstream
    INTEGER sbrng
    INTEGER dbrng
    INTEGER ibrng
```

```
END TYPE VSL_BRNG_PROPERTIES
```

The C version is as follows:

```
typedef struct _VSLBRngProperties {
      int             StreamStateSize;
      int             NSeeds;
      int             IncludesZero;
      int             WordSize;
      int             NBits;
      InitStreamPtr   InitStream;
      sBRngPtr        sBRng;
      dBRngPtr        dBRng;
      iBRngPtr        iBRng;
} VSLBRngProperties;
```

The following table provides brief descriptions of the fields engaged in the above structure:

Table 10-6 Field Descriptions

Field	Short Description
FORTRAN: `streamstatesize` C: `StreamStateSize`	The size, in bytes, of the stream state structure for a given basic generator.
FORTRAN: `nseeds` C: `NSeeds`	The number of 32-bit initial conditions (seeds) necessary to initialize the stream state structure for a given basic generator.
FORTRAN: `includeszero` C: `IncludesZero`	Flag value indicating whether the generator can produce a random 0[1].
FORTRAN: `wordsize` C: `WordSize`	Machine word size, in bytes, used in integer-value computations. Possible values: 4, 8, and 16 for 32, 64, and 128-bit generators, respectively.

Table 10-6 Field Descriptions (continued)

Field	Short Description
FORTRAN: `nbits` C: `NBits`	The number of bits required to represent a random value in integer arithmetic. Note that, for instance, 48-bit random values are stored to 64-bit (8 byte) memory locations. In this case, `WordSize` is equal to 8 (number of bytes used to store the random value), while `NBits` contains the actual number of bits occupied by the value (in this example, `48`).
FORTRAN: `initstream` C: `InitStream`	Contains the pointer to the initialization subroutine of a given basic generator.
FORTRAN: `sbrng` C: `sBRng`	Contains the pointer to the basic generator of single precision real numbers uniformly distributed over the interval (*a,b*) (`REAL` in FORTRAN and `float` in C).
FORTRAN: `dbrng` C: `dBRng`	Contains the pointer to the basic generator of double precision real numbers uniformly distributed over the interval (*a,b*) (`DOUBLE PRECISION` in FORTRAN and `double` in C).
FORTRAN: `ibrng` C: `iBRng`	Contains the pointer to the basic generator of integer numbers with uniform bit distribution[2] (`INTEGER` in FORTRAN and `unsigned int` in C).

1. Certain types of generators, for example, generalized feedback shift registers can potentially generate a random 0. On the other hand, generators like multiplicative congruential generators never generate such a number. In most cases this information is irrelevant because the chance of generating a zero value is small. However, in certain non-uniform distribution generators the possibility for a basic generator to produce a random zero may lead to generation of an infinitely large number (overflow). Even though the software handles overflows correctly, so that they may be interpreted as $+\infty$ and $-\infty$, the user has to be careful and verify the final results. If an infinitely large number may affect the computation, the user should either remove such numbers from the generated vector, or use safe generators, which do not produce random 0.
2. A specific generator that permits operations over single bits and bit groups of random numbers.

RegisterBrng

Registers user-defined basic generator.

Syntax

Fortran:

brng = vslregisterbrng(*properties*)

C:

brng = vslRegisterBrng(*properties*)

Description

An example of a registration procedure can be found in the respective directory of VSL examples.

Input Parameters

FORTRAN:

properties	`TYPE (VSL_BRNG_PROPERTIES), INTENT(IN)`. Structure containing properties of the basic generator to be registered.

C:

properties	`VSLBrngProperties*`. Structure containing properties of the basic generator to be registered.

Output Parameters

FORTRAN:

brng	`INTEGER`. The number (index) of the registered basic generator; used for identification. Negative values indicate the registration error.

C:

brng — `int`. The number (index) of the registered basic generator; used for identification. Negative values indicate the registration error.

GetBrngProperties

Returns structure with properties of a given basic generator.

Syntax

Fortran:

```
call vslgetbrngproperties( brng, properties )
```

C:

```
call vslGetBrngProperties( brng, properties )
```

Input Parameters

FORTRAN:

brng — `INTEGER, INTENT(IN)`. Number (index) of the registered basic generator.

C:

brng — `int`. Number (index) of the registered basic generator.

Output Parameters

FORTRAN:

properties — `TYPE (VSL_BRNG_PROPERTIES), INTENT(OUT)`. Structure containing properties of the generator with number *brng*.

C:

properties — `VSLBrngProperties*`. Structure containing properties of the generator with number *brng*.

Formats for User-Designed Generators

To register a user-designed basic generator using `RegisterBrng` function, you need to pass the pointer *iBrng* to the integer-value implementation of the generator; the pointers *sBrng* and *dBrng* to the generator implementations for single and double precision values, respectively; and pass the pointer *InitStream* to the stream initialization subroutine. This section contains recommendations on defining such functions with input and output arguments. An example of the registration procedure for a user-designed generator can be found in the respective directory of VSL examples.

The respective pointers are defined as follows:

```
typedef   int (*InitStreamPtr)( int method, void * stream, int n,
                                const unsigned int params[] );
typedef   void (*sBRngPtr)( void * stream, int n, float r[],
                            float a, float b );
typedef   void (*dBRngPtr)( void * stream, int n, double r[],
                            double a, double b );
typedef   void (*iBRngPtr)( void * stream, int n,
                            unsigned int r[] );
```

InitStream

FORTRAN:

```
INTEGER FUNCTION mybrnginitstream( method, stream, n, params )
      INTEGER, INTENT (IN) :: method
      TYPE(MYSTREAM_STATE), INTENT (INOUT):: stream
      INTEGER, INTENT (IN) :: n
      INTEGER, INTENT (IN) :: params
! Initialize the stream
      …
```

```
END SUBROUTINE mybrnginitstream
```

C:

```
int MyBrngInitStream( int method, VSLStreamStatePtr stream,
      int n, const unsigned int params[] )
{
       /* Initialize the stream */
       …
} /* MyBrngInitStream */
```

Description

The initialization subroutine of a user-designed generator must initialize *stream* according to the specified initialization *method*, initial conditions *params* and the argument *n*. The value of *method* determines the initialization method to be used.

- If *method* is equal to 0, the initialization is by the standard generation method, which must be supported by all basic generators. In this case the function assumes that the *stream* structure was not previously initialized. The value of *n* is used as the actual number of 32-bit values passed as initial conditions through *params*. Note, that the situation when the actual number of initial conditions passed to the function is not sufficient to initialize the generator is not an error. Whenever it occurs, the basic generator must initialize the missing conditions using default settings.
- If *method* is equal to 1, the generation is by the leapfrog method, where *n* specifies the number of computational nodes (independent streams). Here the function assumes that the *stream* was previously initialized by the standard generation method. In this case *params* contains only one element, which identifies the computational node. If the generator does not support the leapfrog method, the function must return the error code `VSL_ERROR_LEAPFROG_UNSUPPORTED`.
- If *method* is equal to 2, the generation is by the block-splitting method. Same as above, the *stream* is assumed to be previously initialized by the standard generation method; *params* is not used, *n* identifies the number of skipped elements. If the generator does not support the block-splitting method, the function must return the error code `VSL_ERROR_SKIPAHEAD_UNSUPPORTED`.

For a more detailed description of the leapfrog and the block-splitting methods, refer to the description of `LeapfrogStream` and `SkipAheadStream`, respectively.

Stream state structure is individual for every generator. However, each structure has a number of fields that are the same for all the generators:

FORTRAN:

```
type(mystream_state)
     INTEGER*4     reserved1
     INTEGER*4     reserved2
     INTEGER*4     reserved3
     INTEGER*4     reserved4
     [ fields specific for the given generator ]
end type mystream_state
```

C:

```
typedef struct
{
     uint64Reserved1;
     uint64Reserved2;
     [ fields specific for the given generator ]
} MyStreamState
```

The fields `Reserved1` and `Reserved2` are reserved for private needs only, and must not be modified by the user. When including specific fields into the structure, follow the rules below:

- The fields must fully describe the current state of the generator. For example, the state of a linear congruential generator can be identified by only one initial condition;
- If the generator can use both the leapfrog and the block-splitting methods, additional fields should be introduced to identify the independent streams. For example, in $LCG(a, c, m)$, apart from the initial conditions, two more fields should be specified: the value of the multiplier a^k and the value of the increment $(a^k - 1)c/(a - 1)$.

For a more detailed discussion, refer to [Knuth81], and [Gentle98]. An example of the registration procedure can be found in the respective directory of VSL examples.

iBRng

FORTRAN:

```
SUBROUTINE imybrng( stream, n, r )
```

```
      TYPE(MYSTREAM_STATE), INTENT(INOUT):: stream
      INTEGER, INTENT(IN)   :: n
      INTEGER, DIMENSION(*), INTENT(OUT) :: r
! Generating integer random numbers
! Pay attention to word size needed to
! store one random number
      DO i = 1, n
             R(I) = …
      END DO
! Update stream state
END SUBROUTINE imybrng
```

C:

```
void iMyBrng( VSLStreamStatePtr stream, int n,
              unsigned int r[] )
{
       int       i;    /* Loop variable */
      /* Generating integer random numbers */
      /* Pay attention to word size needed to
         store only random number */
      for( i = 0; i < n; i++ )
      {
                r[i] = …
      }
      /* Update stream state */
      …
} /* iMyBrng */
```

NOTE. When using 64 and 128-bit generators, consider digit capacity to store the numbers to the random vector *r* correctly. For example, storing one 64-bit value requires two elements of *r* , the first to store the lower 32 bits and the second to store the higher 32 bits. Similarly, use 4 elements of *r* to store a 128-bit value.

sBRng

FORTRAN:

```
SUBROUTINE smybrng( stream, n, r, a, b )
      TYPE(MYSTREAM_STATE), INTENT(INOUT):: stream
      INTEGER, INTENT(IN)   :: n
      REAL, DIMENSION(n), INTENT(OUT)   :: r
      REAL, INTENT(IN)   :: a
      REAL, INTENT(IN)   :: b
! Generating real (a,b) random numbers
      DO i = 1, n
            R(I) = …
      END DO
! Update stream state
END SUBROUTINE smybrng
```

C:

```
void sMyBrng( VSLStreamStatePtr stream, int n, float r[],
              float a, float b )
{
       int       i;    /* Loop variable */
      /* Generating float (a,b) random numbers */
      for ( i = 0; i < n; i++ )
      {
                r[i] = …
      }
```

```
      /* Update stream state */
      …
} /* sMyBrng */
```

dBRng

FORTRAN:

```
SUBROUTINE dmybrng( stream, n, r, a, b )
      TYPE(MYSTREAM_STATE), INTENT(INOUT) :: stream
      INTEGER, INTENT(IN)    :: n
      DOUBLE PRECISION, DIMENSION(n), INTENT(OUT)  :: r
      REAL, INTENT(IN)  :: a
      REAL, INTENT(IN)  :: b
! Generating double precision (a,b) random numbers
      DO i = 1, n
            R(I) = …
      END DO
! Update stream state
             …
END SUBROUTINE dmybrng
```

C:

```
void dMyBrng( VSLStreamStatePtr stream, int n, double r[],
              double a, double b )
{
       int      i;    /* Loop variable */
      /* Generating double (a,b) random numbers */
      for ( i = 0; i < n; i++ )
      {
                r[i] = …
      }
      /* Update stream state */
      …
} /* dMyBrng */
```

Discrete Fourier Transform Functions

11

This chapter describes the set of Discrete Fourier ransform (DFT) functions implemented in Intel MKL, which present a uniform and easy-to-use Applications Programmer Interface providing fast computation of DFT via the Fast Fourier Transform (FFT) algorithm.

The Discrete Fourier Transform function library of Intel MKL provides one-dimensional, two-dimensional, and multi-dimensional (up to the order of 7) routines and both Fortran- and C-interfaces for all transform functions.

For compatibility with previous versions, Intel MKL still supports the older FFT interface described in chapter 12 of this manual, but users of this code are encouraged to migrate to the new advanced DFT functions in their application programs for both performance and flexibility. Unlike the older FFT routines, the DFT functions support transform lengths of other than powers of 2 mixed radix.

The full list of DFT functions implemented in Intel MKL is given in the table below:

Table 11-1 DFT Functions in Intel MKL

Function Name	Operation
Descriptor Manipulation Functions	
`DftiCreateDescriptor`	Allocates memory for the descriptor data structure and instantiates it with default configuration settings.
`DftiCommitDescriptor`	Performs all initialization that facilitates the actual DFT computation.
`DftiCopyDescriptor`	Copies an existing descriptor.
`DftiFreeDescriptor`	Frees memory allocated for a descriptor.
DFT Computation Functions	
`DftiComputeForward`	Computes the forward DFT.
`DftiComputeBackward`	Computes the backward DFT.

Table 11-1 DFT Functions in Intel MKL (continued)

Function Name	Operation
Descriptor Configuration Functions	
`DftiSetValue`	Sets one particular configuration parameter with the specified configuration value.
`DftiGetValue`	Gets the configuration value of one particular configuration parameter.
Status Checking Functions	
`DftiErrorClass`	Checks if the status reflects an error of a predefined class.
`DftiErrorMessage`	Generates an error message.

Description of DFT functions is followed by discussion of configuration settings (see Configuration Settings) and various configuration parameters used.

Computing DFT

DFT functions described later in this chapter are implemented in Fortran and C interface. Fortran stands for Fortran 95. DFT interface relies critically on many modern features offered in Fortran 95 that have no counterpart in Fortran 77

NOTE. Following the explicit function interface in Fortran, data array must be defined as one-dimensional for any transformation type.

The materials presented in this chapter assume the availability of native complex types in C as they are specified in C9X.

You can find example code that uses DFT interface functions to compute transform results in "DFT Code Examples" section in the appendix.

For most common situations, we expect a DFT computation can be effected by four function calls. The approach adopted in Intel MKL for DFT computation uses one single data structure, the descriptor, to record flexible configuration whose parameters can be changed independently. This results in enhanced functionality and ease of use.

The record of type `DFTI_DESCRIPTOR`, when created, contains information about the length and domain of the DFT to be computed, as well as the setting of a rather large number of configuration parameters. The default settings for all of these parameters include, for example, the following:

- the DFT to be computed does not have a scale factor;
- there is only one set of data to be transformed;
- the data is stored contiguously in memory;
- the forward transform is defined to be the formula using $e^{-i2\pi jk/n}$ rather than $e^{+i2\pi jk/n}$;
- complex data is stored in the native complex data type;
- the computed result overwrites (in place) the input data; etc.

Should any one of these many default settings be inappropriate, they can be changed one-at-a-time through the function `DftiSetValue` as illustrated in the Example C-17 and Example C-18.

DFT Interface

To use the DFT functions, you need to access the module `MKL_DFTI` through the "`use`" statement in Fortran; or access the header file `mkl_dfti.h` through "`include`" in C.

The Fortran interface provides a derived type `DFTI_DESCRIPTOR`; a number of named constants representing various names of configuration parameters and their possible values; and a number of overloaded functions through the generic functionality of Fortran 95.

The C interface provides a structure type `DFTI_DESCRIPTOR`, a macro definition

```
#define DFTI_DESCRIPTOR_HANDLE DFTI_DESCRIPTOR *;
```

a number of named constants of two enumeration types `DFTI_CONFIG_PARAM` and `DFTI_CONFIG_VALUE`;
and a number of functions, some of which accept different number of input arguments.

NOTE. Some of the functions and/or functionality described in the subsequent sections of this chapter may not be supported by the currently available implementation of the library. You can find the complete list of the implementation-specific exceptions in the release notes to your version of the library.

There are four main categories of DFT functions in Intel MKL:

1. **Descriptor Manipulation.** There are four functions in this category. The first one, `DftiCreateDescriptor`, creates a DFT descriptor whose storage is allocated dynamically by the routine. This function configures the descriptor with default settings corresponding to a few input values supplied by the user.

   The second, `DftiCommitDescriptor`, "commits" the descriptor to all its setting. In practice, this usually means that all the necessary precomputation will be performed. This may include factorization of the input length and computation of all the required twiddle factors. The third function, `DftiCopyDescriptor`, makes an extra copy of a descriptor, and the fourth function, `DftiFreeDescriptor`, frees up all the memory allocated for the descriptor information.

2. **DFT Computation**. There are two functions in this category. The first, `DftiComputeForward`, effects a forward DFT computation, and the second function, `DftiComputeBackward`, performs a backward DFT computation.

3. **Descriptor configuration**. There are two functions in this category. One function, `DftiSetValue`, sets one specific value to one of the many configuration parameters that are changeable (a few are not); the other, `DftiGetValue`, gets the current value of any one of these configuration parameters (all are readable). These parameters, though many, are handled one-at-a-time.

4. **Status Checking**. The functions described in the three categories above return an integer value denoting the status of the operation.
   In particular, a non-zero return value always indicates a problem of some sort. Envisioned to be further enhanced in later releases of Intel MKL, DFT interface at present provides for one logical status class function, `DftiErrorClass`, and a simple status message generation function, `DftiErrorMessage`.

Status Checking Functions

All of the descriptor manipulation, DFT computation, and descriptor configuration functions return an integer value denoting the status of the operation. Two functions serve to check the status. The first function is a logical function that checks if the status reflects an error of a predefined class, and the second is an error message function that returns a character string.

ErrorClass

Checks if the status reflects an error of a predefined class.

Syntax

```
! Fortran
Predicate = DftiErrorClass( Status, Error_Class )
/* C */
predicate = DftiErrorClass( status, error_class );
```

Description

DFT interface in Intel MKL provides a set of predefined error class listed in Table 11-2. These are named constants and have the type `INTEGER` in Fortran and `long` in C.

Table 11-2 Predefined Error Class

Named Constants	Comments
`DFTI_NO_ERROR`	No error
`DFTI_MEMORY_ERROR`	Usually associated with memory allocation
`DFTI_INVALID_CONFIGURATION`	Invalid settings of one or more configuration parameters
`DFTI_INCONSISTENT_CONFIGURATION`	Inconsistent configuration or input parameters
`DFTI_NUMBER_OF_THREADS_ERROR`	Number of OMP threads in the computation function is not equal to the number of OMP threads in the initialization stage (commit function)
`DFTI_MULTITHREADED_ERROR`	Usually associated with OMP routine's error return value
`DFTI_BAD_DESCRIPTOR`	Descriptor is unusable for computation
`DFTI_UNIMPLEMENTED`	Unimplemented legitimate settings; implementation dependent
`DFTI_MKL_INTERNAL_ERROR`	Internal library error

Note that the correct usage is to check if the status returns `.TRUE.` or `.FALSE.` through the use of `DFTI_ERROR_CLASS` with a specific error class. Direct comparison of a status with the predefined class is an incorrect usage. See Example C-19 on a correct use of the status checking functions.

Interface and prototype

```
//Fortran interface
INTERFACE DftiErrorClass
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
 FUNCTION some_actual_function_8( Status, Error_Class )
   LOGICAL some_actual_function_8
   INTEGER, INTENT(IN) :: Status, Error_Class
 END FUNCTION some_actual_function_8
END INTERFACE DftiErrorClass

/* C prototype */
long DftiErrorClass( long , long );
```

ErrorMessage

Generates an error message.

Syntax

```
! Fortran
ERROR_MESSAGE = DftiErrorMessage( Status )
/* C */
error_message = DftiErrorMessage( status );
```

Description

The error message function generates an error message character string. The maximum length of the string in Fortran is given by the named constant `DFTI_MAX_MESSAGE_LENGTH`. The actual error message is implementation dependent. In Fortran, the user needs to use a character string of length `DFTI_MAX_MESSAGE_LENGTH` as the target. In C, the function returns a pointer to a character string, that is, a character array with the delimiter ' 0'.

Example C-19 shows how this function can be implemented.

Interface and prototype

```
//Fortran interface
INTERFACE DftiErrorMessage
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
 FUNCTION some_actual_function_9( Status, Error_Class )
   CHARACTER(LEN=DFTI_MAX_MESSAGE_LENGTH) some_actual_function_9( Status )
   INTEGER, INTENT(IN) :: Status
 END FUNCTION some_actual_function_9
END INTERFACE DftiErrorMessage

/* C prototype */
char *DftiErrorMessage( long );
```

Descriptor Manipulation

There are four functions in this category: create a descriptor, commit a descriptor, copy a descriptor, and free a descriptor.

CreateDescriptor

Allocates memory for the descriptor data structure and instantiates it with default configuration settings.

Syntax

```
! Fortran
```

```
     Status = DftiCreateDescriptor( Desc_Handle,      &
                                    Precision,        &
                                    Forward_Domain,   &
                                    Dimension,        &
                                    Length )

/* C */
     status = DftiCreateDescriptor( &desc_handle,
                                    precision,
                                    forward_domain,
                                    dimension,
                                    length );
```

Description

This function allocates memory for the descriptor data structure and instantiates it with all the default configuration settings with respect to the precision, domain, dimension, and length of the desired transform. The domain is understood to be the domain of the forward transform. Since memory is allocated dynamically, the result is actually a pointer to the created descriptor. This function is slightly different from the "initialization" routine in more traditional software packages or libraries used for computing DFT. In all likelihood, this function will not perform any significant computation work such as twiddle factors computation, as the default configuration settings can still be changed upon user's request through the value setting function `DftiSetValue`.

The precision and (forward) domain are specified through named constants provided in DFT interface for the configuration values. The choices for precision are `DFTI_SINGLE` and `DFTI_DOUBLE`; and the choices for (forward) domain are `DFTI_COMPLEX`, `DFTI_REAL`, and `DFTI_CONJUGATE_EVEN`. See Table 11-5 for the complete table of named constants for configuration values.

Dimension is a simple positive integer indicating the dimension of the transform. Length is either a simple positive integer for one-dimensional transform, or an integer array (pointer in C) containing the positive integers corresponding to the lengths dimensions for multi-dimensional transform.

The function returns `DFTI_NO_ERROR` when completes successfully. See Status Checking Functions for more information on returned status.

Interface and prototype

```
!Fortran interface.
INTERFACE DftiCreateDescriptor
```

```
!Note that the body provided here is to illustrate the different
!argument list and types of dummy arguments. The interface
!does not guarantee what the actual function names are.
!Users can only rely on the function name following the keyword INTERFACE
 FUNCTION some_actual_function_1D( Desc_Handle, Prec, Dom, Dim, Length )
   INTEGER :: some_actual_function_1D
   TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
   INTEGER, INTENT(IN) :: Prec, Dom
   INTEGER, INTENT(IN) :: Dim, Length
 END FUNCTION some_actual_function_1D

 FUNCTION some_actual_function_HIGHD( Desc_Handle, Prec, Dom, Dim, Length )
   INTEGER :: some_actual_function_HIGHD
   TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
   INTEGER, INTENT(IN) :: Prec, Dom
   INTEGER, INTENT(IN) :: Dim, Length(*)
 END FUNCTION some_actual_function_HIGHD
END INTERFACE DftiCreateDescriptor
```

Note that the function is overloaded as the actual argument for `Length` can be a scalar or a a rank-one array.

```
/* C prototype */
long DftiCreateDescriptor( DFTI_DESCRIPTOR_HANDLE *,
                           DFTI_CONFIG_PARAM ,
                           DFTI_CONFIG_PARAM ,
                           long ,
                           ... );
```

The variable arguments facility is used to cope with the argument for lengths that can be a scalar (`long`), or an array (`long *`).

CommitDescriptor

Performs all initialization that facilitates the actual DFT computation.

Syntax

```
! Fortran
Status = DftiCommitDescriptor( Desc_Handle )
/* C */
status = DftiCommitDescriptor( desc_handle );
```

Description

The interface requires a function that commits a previously created descriptor be invoked before the descriptor can be used for DFT computations. Typically, this committal performs all initialization that facilitates the actual DFT computation. For a modern implementation, it may involve exploring many different factorizations of the input length to search for highly efficient computation method.

Any changes of configuration parameters of a committed descriptor via the set value function (see Descriptor Configuration) requires a re-committal of the descriptor before a computation function can be invoked. Typically, this committal function call is immediately followed by a computation function call (see DFT Computation).

The function returns `DFTI_NO_ERROR` when completes successfully. See Status Checking Functions for more information on returned status.

Interface and prototype

```
! Fortran interface
INTERFACE DftiCommitDescriptor
!Note that the body provided here is to illustrate the different
!argument list and types of dummy arguments. The interface
!does not guarantee what the actual function names are.
!Users can only rely on the function name following the
!keyword INTERFACE
 FUNCTION some_actual function_1 ( Desc_Handle )
   INTEGER :: some_actual function_1
```

```
   TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 END FUNCTION some_actual function_1
END INTERFACE DftiCommitDescriptor

/* C prototype */
long DftiCommitDescriptor( DFTI_DESCRIPTOR_HANDLE );
```

CopyDescriptor

Copies an existing descriptor.

Syntax

```
! Fortran
Status = DftiCopyDescriptor( Desc_Handle_Original,
                             Desc_Handle_Copy )
/* C */
status = DftiCopyDescriptor( desc_handle_original,
                             &desc_handle_copy );
```

Description

This function makes a copy of an existing descriptor and provides a pointer to it. The purpose is that all information of the original descriptor will be maintained even if the original is destroyed via the free descriptor function `DftiFreeDescriptor`.

The function returns `DFTI_NO_ERROR` when completes successfully. See Status Checking Functions for more information on returned status.

Interface and prototype

```
! Fortran interface
INTERFACE DftiCopyDescriptor
! Note that the body provided here is to illustrate the different
!argument list and types of dummy arguments. The interface
!does not guarantee what the actual function names are.
!Users can only rely on the function name following the
!keyword INTERFACE
```

```
 FUNCTION some_actual_function_2( Desc_Handle_Original,
                                   Desc_Handle_Copy )
   INTEGER :: some_actual_function_2
   TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle_Original, Desc_Handle_Copy
 END FUNCTION some_actual_function_2
END INTERFACE DftiCopyDescriptor

/* C prototype */
long DftiCopyDescriptor( DFTI_DESCRIPTOR_HANLDE, DFTI_DESCRIPTOR_HANDLE * );
```

FreeDescriptor

Frees memory allocated for a descriptor.

Syntax

```
! Fortran
Status = DftiFreeDescriptor( Desc_Handle )
/* C */
status = DftiFreeDescriptor( &desc_handle );
```

Description

This function frees up all memory space allocated for a descriptor.

The function returns `DFTI_NO_ERROR` when completes successfully. See Status Checking Functions for more information on returned status.

Interface and prototype

```
! Fortran interface
INTERFACE DftiFreeDescriptor
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
```

```
 FUNCTION some_actual_function_3( Desc_Handle )
   INTEGER :: some_actual_function_3
   TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 END FUNCTION some_actual_function_3
END INTERFACE DftiFreeDescriptor

/* C prototype */
long DftiFreeDescriptor( DFTI_DESCRIPTOR_HANDLE * );
```

DFT Computation

There are two functions in this category: compute the forward transform, and compute the backward transform.

ComputeForward

Computes the forward DFT.

Syntax

```
! Fortran
Status = DftiComputeForward( Desc_Handle, X_inout )
Status = DftiComputeForward( Desc_Handle, X_in, X_out )
Status = DftiComputeForward( Desc_Handle, X_inout, Y_inout )
Status = DftiComputeForward( Desc_Handle, X_in, Y_in, X_out, Y_out )
/* C */
status = DftiComputeForward( desc_handle, x_inout );
status = DftiComputeForward( desc_handle, x_in, x_out );
status = DftiComputeForward( desc_handle, x_inout, y_inout );
status = DftiComputeForward( desc_handle, x_in, y_in, x_out, y_out );
```

Description

As soon as a descriptor is configured and committed successfully, actual computation of DFT can be performed. The `DftiComputeForward` function computes the forward DFT. By default, this is the transform using the factor $e^{-i2\pi/n}$ (instead of the one with a positive sign). Because of the flexibility in configuration, input data can be represented in various ways as well as output result can be placed differently. Consequently, the number of input parameters as well as their type vary. This variation is accommodated by the generic function facility of Fortran 95. Data and result parameters are all declared as assumed-size rank-1 array `DIMENSION(0:*)`.

The function returns `DFTI_NO_ERROR` when completes successfully. See

Status Checking Functions for more information on returned status.

Interface and prototype

```
//Fortran interface.
INTERFACE DftiComputeFoward
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
 // One argument single precision complex
 FUNCTION some_actual_function_4_C( Desc_Handle, X )
   INTEGER :: some_actual_function_4_C
   TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
   COMPLEX, INTENT(INOUT) :: X(*)
 END FUNCTION some_actual_function_4_C
 // One argument double precision complex
 FUNCTION some_actual_function_4_Z( Desc_Handle, X )
   INTEGER :: some_actual_function_4_Z
   TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
   COMPLEX (Kind((0D0,0D0))), INTENT(INOUT) :: X(*)
 END FUNCTION some_actual_function_4_Z
 // One argument single precision real
 FUNCTION some_actual_function_4_R( Desc_Handle, X )
```

```
  INTEGER :: some_actual_function_4_R
  TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
  REAL, INTENT(INOUT) :: X(*)
 END FUNCTION some_actual_function_4_R
 // One argument double precision real
 ...
 // Two argument single precision complex
 ...
 ...
 // Four argument double precision real
 FUNCTION some_actual_function_4_DDDD( Desc_Handle, X1_In, X2_In,
                                        Y1_Out, Y2_Out )
  INTEGER :: some_actual_function_4_DDDD
  TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
  REAL (Kind(0D0)), INTENT(IN) :: X1_In(*), X2_In(*)
  REAL (Kind(0D0)), INTENT(OUT) :: Y1_Out(*), Y2_Out(*)
 END FUNCTION some_actual_function_4_DDDD
END INTERFACE DftiComputeFoward

/* C prototype */
long DftiComputeForward( DFTI_DESCRIPTOR_HANDLE,
                         void *,
                           ... );
```

The implementations of DFT interface expect the data be treated as data stored linearly in memory with a regular "stride" pattern (discussed more fully in Strides, see also [3]). The function expects the starting address of the first element. Hence we use the assume-size declaration in Fortran.

The descriptor by itself contains sufficient information to determine exactly how many arguments and of what type should be present. The implementation could use this information to check against possible input inconsistency.

ComputeBackward

Computes the backward DFT.

Syntax

```
! Fortran
Status = DftiComputeBackward( Desc_Handle, X_inout )
Status = DftiComputeBackward( Desc_Handle, X_in, X_out )
Status = DftiComputeBackward( Desc_Handle, X_inout, Y_inout )
Status = DftiComputeBackward( Desc_Handle, X_in, Y_in, X_out, Y_out )
/* C */
status = DftiComputeBackward( desc_handle, x_inout );
status = DftiComputeBackward( desc_handle, x_in, x_out );
status = DftiComputeBackward( desc_handle, x_inout, y_inout );
status = DftiComputeBackward( desc_handle, x_in, y_in, x_out, y_out );
```

Description

As soon as a descriptor is configured and committed successfully, actual computation of DFT can be performed. The `DftiComputeBackward` function computes the backward DFT.

By default, this is the transform using the factor $e^{i2\pi/n}$ (instead of the one with a negative sign). Because of the flexibility in configuration, input data can be represented in various ways as well as output result can be placed differently. Consequently, the number of input parameters as well as their type vary. This variation is accommodated by the generic function facility of Fortran 95. Data and result parameters are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. The function returns `DFTI_NO_ERROR` when completes successfully. See Status Checking Functions for more information on returned status.

Interface and prototype

```
//Fortran interface.
INTERFACE DftiComputeBackward
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
```

```
//keyword INTERFACE
 // One argument single precision complex
 FUNCTION some_actual_function_5_C( Desc_Handle, X )
   INTEGER :: some_actual_function_5_C
   TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
   COMPLEX, INTENT(INOUT) :: X(*)
 END FUNCTION some_actual_function_5_C
 // One argument double precision complex
 FUNCTION some_actual_function_5_Z( Desc_Handle, X )
   INTEGER :: some_actual_function_5_Z
   TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
   COMPLEX (Kind((0D0,0D0))), INTENT(INOUT) :: X(*)
 END FUNCTION some_actual_function_5_Z
 // One argument single precision real
 FUNCTION some_actual_function_5_R( Desc_Handle, X )
   INTEGER :: some_actual_function_5_R
   TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
   REAL, INTENT(INOUT) :: X(*)
 END FUNCTION some_actual_function_5_R
 // One argument double precision real
 ...
 // Two argument single precision complex
 ...
 ...
 // Four argument double precision real
 FUNCTION some_actual_function_5_DDDD( Desc_Handle, X1_In, X2_In,
                                       Y1_Out, Y2_Out )
   INTEGER :: some_actual_function_5_DDDD
   TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
   REAL (Kind(0D0)), INTENT(IN) :: X1_In(*), X2_In(*)
   REAL (Kind(0D0)), INTENT(OUT) :: Y1_Out(*), Y2_Out(*)
 END FUNCTION some_actual_function_5_DDDD
```

```
END INTERFACE DftiComputeBackward
```

```
/* C prototype */
long DftiComputeBackward( DFTI_DESCRIPTOR_HANDLE,
                          void *,
                           ... );
```

The implementations of DFT interface expect the data be treated as data stored linearly in memory with a regular "stride" pattern (discussed more fully in Strides, see also [3]). The function expects the starting address of the first element. Hence we use the assume-size declaration in Fortran.

The descriptor by itself contains sufficient information to determine exactly how many arguments and of what type should be present. The implementation could use this information to check against possible input inconsistency.

Descriptor Configuration

There are two functions in this category: the value setting function `DftiSetValue` sets one particular configuration parameter to an appropriate value, and the value getting function `DftiGetValue` reads the values of one particular configuration parameter. While all configuration parameters are readable, a few of them cannot be set by user. Some of these contain fixed information of a particular implementation such as version number, or dynamic information, but nevertheless are derived by the implementation during execution of one of the functions. See Configuration Settings for details.

SetValue

Sets one particular configuration parameter with the specified configuration value.

Syntax

```
! Fortran
Status = DftiSetValue( Desc_Handle, &
                       Config_Param, &
                       Config_Val )
```

```
/* C */
```

```
status = DftiSetValue( desc_handle,
                        config_param,
                          config_val );
```

Description

This function sets one particular configuration parameter with the specified configuration value. The configuration parameter is one of the named constants listed in Table 11-3, and the configuration value is the corresponding appropriate type, which can be a named constant or a native type. See Configuration Settings for details of the meaning of the setting.

The function returns `DFTI_NO_ERROR` when completes successfully. See Status Checking Functions for more information on returned status.

Interface and prototype

```
! Fortran interface
INTERFACE DftiSetValue
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
 FUNCTION some_actual_function_6_INTVAL( Desc_Handle, Config_Param, INTVAL )
   INTEGER :: some_actual_function_6_INTVAL
   Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
   INTEGER, INTENT(IN) :: Config_Param
   INTEGER, INTENT(IN) :: INTVAL
 END FUNCTION some_actual_function_6_INTVAL

 FUNCTION some_actual_function_6_SGLVAL( Desc_Handle, Config_Param, SGLVAL )
   INTEGER :: some_actual_function_6_SGLVAL
   Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
   INTEGER, INTENT(IN) :: Config_Param
   REAL, INTENT(IN) :: SGLVAL
END FUNCTION some_actual_function_6_SGLVAL
```

```
 FUNCTION some_actual_function_6_DBLVAL( Desc_Handle, Config_Param, DBLVAL )
   INTEGER :: some_actual_function_6_DBLVAL
   Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
   INTEGER, INTENT(IN) :: Config_Param
   REAL (KIND(0D0)), INTENT(IN) :: DBLVAL
 END FUNCTION some_actual_function_6_DBLVAL

 FUNCTION some_actual_function_6_INTVEC( Desc_Handle, Config_Param, INTVEC )
   INTEGER :: some_actual_function_6_INTVEC
   Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
   INTEGER, INTENT(IN) :: Config_Param
   INTEGER, INTENT(IN) :: INTVEC(*)
 END FUNCTION some_actual_function_6_INTVEC

 FUNCTION some_actual_function_6_CHARS( Desc_Handle, Config_Param, CHARS )
   INTEGER :: some_actual_function_6_CHARS
   Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
   INTEGER, INTENT(IN) :: Config_Param
   CHARCTER(*), INTENT(IN) :: CHARS
 END FUNCTION some_actual_function_6_CHARS
END INTERFACE DftiSetValue


/* C prototype */
long DftiSetValue( DFTI_DESCRIPTOR_HANDLE,
                   DFTI_CONFIG_PARAM ,
                    ... );
```

GetValue

Gets the configuration value of one particular configuration parameter.

Syntax

```
! Fortran
     Status = DftiGetValue( Desc_Handle,      &
                            Config_Param,     &
                            Config_Val )
/* C */
     status = DftiGetValue( desc_handle,
                            config_param,
                            &config_val );
```

Description

This function gets the configuration value of one particular configuration parameter. The configuration parameter is one of the named constants listed in Table 11-3 and Table 11-4, and the configuration value is the corresponding appropriate type, which can be a named constant or a native type.

The function returns `DFTI_NO_ERROR` when completes successfully. See Status Checking Functions for more information on returned status.

Interface and prototype

```
! Fortran interface
INTERFACE DftiGetValue
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
 FUNCTION some_actual_function_7_INTVAL( Desc_Handle, Config_Param, INTVAL )
   INTEGER :: some_actual_function_7_INTVAL
   Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
   INTEGER, INTENT(IN) :: Config_Param
```

```
    INTEGER, INTENT(OUT) :: INTVAL
  END FUNCTION DFTI_GET_VALUE_INTVAL

  FUNCTION some_actual_function_7_SGLVAL( Desc_Handle, Config_Param, SGLVAL )
    INTEGER :: some_actual_function_7_SGLVAL
    Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    INTEGER, INTENT(IN) :: Config_Param
    REAL, INTENT(OUT) :: SGLVAL
  END FUNCTION some_actual_function_7_SGLVAL

  FUNCTION some_actual_function_7_DBLVAL( Desc_Handle, Config_Param, DBLVAL )
    INTEGER :: some_actual_function_7_DBLVAL
    Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    INTEGER, INTENT(IN) :: Config_Param
    REAL (KIND(0D0)), INTENT(OUT) :: DBLVAL
  END FUNCTION some_actual_function_7_DBLVAL

  FUNCTION some_actual_function_7_INTVEC( Desc_Handle, Config_Param, INTVEC )
    INTEGER :: some_actual_function_7_INTVEC
    Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    INTEGER, INTENT(IN) :: Config_Param
    INTEGER, INTENT(OUT) :: INTVEC(*)
  END FUNCTION some_actual_function_7_INTVEC

  FUNCTION some_actual_function_7_INTPNT( Desc_Handle, Config_Param, INTPNT )
    INTEGER :: some_actual_function_7_INTPNT
    Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    INTEGER, INTENT(IN) :: Config_Param
    INTEGER, DIMENSION(*), POINTER :: INTPNT
  END FUNCTION some_actual_function_7_INTPNT

  FUNCTION some_actual_function_7_CHARS( Desc_Handle, Config_Param, CHARS )
    INTEGER :: some_actual_function_7_CHARS
    Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
```

```
   INTEGER, INTENT(IN) :: Config_Param
   CHARCTER(*), INTENT(OUT):: CHARS
 END FUNCTION some_actual_function_7_CHARS
END INTERFACE DftiGetValue

/* C prototype */
long DftiGetValue( DFTI_DESCRIPTOR_HANDLE,
                   DFTI_CONFIG_PARAM ,
                    ... );
```

Configuration Settings

Each of the configuration parameters is identified by a named constant in the MKL_DFTI module. In C, these named constants have the enumeration type DFTI_CONFIG_PARAM. The list of configuration parameters whose values can be set by user is given in Table 11-3; the list of configuration parameters that are read-only is given in Table 11-4. All parameters are readable. Most of these parameters are self-explanatory, while some others are discussed more fully in the description of the relevant functions

Table 11-3 Settable Configuration Parameters

Named Constants	Value Type	Comments
Most common configurations, no default, must be set explicitly		
DFTI_PRECISION	Named constant	Precision of computation
DFTI_FORWARD_DOMAIN	Named constant	Domain for the forward transform
DFTI_DIMENSION	Integer scalar	Dimension of the transform
DFTI_LENGTHS	Integer scalar/array	Lengths of each dimension
Common configurations including multiple transform and data representation		
DFTI_NUMBER_OF_TRANSFORMS	Integer scalar	For multiple number of transforms
DFTI_FORWARD_SIGN	Named constant	The definition for forward transform
DFTI_FORWARD_SCALE	Floating-point scalar	Scale factor for forward transform
DFTI_BACKWARD_SCALE	Floating-point scalar	Scale factor for backward transform
DFTI_PLACEMENT	Named constant	Placement of the computation result
DFTI_COMPLEX_STORAGE	Named constant	Storage method, complex domain data
DFTI_REAL_STORAGE	Named constant	Storage method, real domain data
DFTI_CONJUGATE_EVEN_STORAGE	Named constant	Storage method, conjugate even domain data
DFTI_DESCRIPTOR_NAME	Character string	No longer than DFTI_MAX_NAME_LENGTH
DFTI_PACKED_FORMAT	Named constant	Packed format, real domain data
DFTI_NUMBER_OF_USER_THREADS	Integer scalar	Number of user threads employing the same descriptor for DFT computation

Table 11-3 Settable Configuration Parameters (continued)

Named Constants	Value Type	Comments
Configurations regarding stride of data		
DFTI_INPUT_DISTANCE	Integer scalar	Multiple transforms, distance of first elements
DFTI_OUTPUT_DISTANCE	Integer scalar	Multiple transforms, distance of first elements
DFTI_INPUT_STRIDES	Integer array	Stride information of input data
DFTI_OUTPUT_STRIDES	Integer array	Stride information of output data
Advanced configuration		
DFTI_INITIALIZATION_EFFORT	Named constant	Dynamic search for computation method
DFTI_ORDERING	Named constant	Scrambling of data order
DFTI_WORKSPACE	Named constant	Computation without auxiliary storage
DFTI_TRANSPOSE	Named constant	Scrambling of dimension

Table 11-4 Read-Only Configuration Parameters

Named Constants	Value Type	Comments
DFTI_COMMIT_STATUS	Name constant	Whether descriptor has been committed
DFTI_VERSION	String	Intel MKL library version number
DFTI_FORWARD_ORDERING	Integer pointer	Pointer to an integer array (see Ordering)
DFTI_BACKWARD_ORDERING	Integer pointer	Pointer to an integer array (see Ordering)

The configuration parameters are set by various values. Some of these values are specified by native data types such as an integer value (for example, number of simultaneous transforms requested), or a single-precision number (for example, the scale factor one would like to apply on a forward transform).

Other configuration values are discrete in nature (for example, the domain of the forward transform) and are thus provided in the DFTI module as named constants. In C, these named constants have the enumeration type `DFTI_CONFIG_VALUE`. The complete list of named constants used for this kind of configuration values is given in Table 11-5.

Table 11-5 Named Constant Configuration Values

Named Constant	Comments
`DFTI_SINGLE`	Single precision
`DFTI_DOUBLE`	Double precision
`DFTI_COMPLEX`	Complex domain
`DFTI_REAL`	Real domain
`DFTI_CONJUGATE_EVEN`	Conjugate even domain
`DFTI_NEGATIVE`	Sign used to define the forward transform
`DFTI_POSITIVE`	Sign used to define the forward transform
`DFTI_INPLACE`	Output overwrites input
`DFTI_NOT_INPLACE`	Output does not overwrite input
`DFTI_COMPLEX_COMPLEX`	Storage method (see Storage schemes)
`DFTI_REAL_REAL`	Storage method (see Storage schemes)
`DFTI_COMPLEX_REAL`	Storage method (see Storage schemes)
`DFTI_REAL_COMPLEX`	Storage method (see Storage schemes)
`DFTI_HIGH`	A high setting, related to initialization effort
`DFTI_MEDIUM`	A medium setting, related to initialization effort
`DFTI_LOW`	A low setting, related to initialization effort
`DFTI_COMMITTED`	Committal status of a descriptor
`DFTI_UNCOMMITTED`	Committal status of a descriptor
`DFTI_ORDERED`	Data ordered in both forward and backward domains
`DFTI_BACKWARD_SCRAMBLED`	Data scrambled in backward domain (by forward transform)
`DFTI_FORWARD_SCRAMBLED`	Data scrambled in forward domain (by backward transform)
`DFTI_ALLOW`	Allow certain request or usage if useful
`DFTI_AVOID`	Avoid certain request or usage if practical
`DFTI_NONE`	Used to specify no transposition
`DFTI_CCS_FORMAT`	Packed format, real data (see "Packed formats")
`DFTI_PACK_FORMAT`	Packed format, real data (see "Packed formats")
`DFTI_PERM_FORMAT`	Packed format, real data (see "Packed formats")

Table 11-5 Named Constant Configuration Values (continued)

Named Constant	Comments
DFTI_VERSION_LENGTH	Number of characters for library version length
DFTI_MAX_NAME_LENGTH	Maximum descriptor name length
DFTI_MAX_MESSAGE_LENGTH	Maximum status message length

Table 11-6 lists the possible values for those configuration parameters that are discrete in nature.

Table 11-6 Settings for Discrete Configuration Parameters

Named Constant	Possible Values
DFTI_PRECISION	DFTI_SINGLE, or
	DFTI_DOUBLE (no default)
DFTI_FORWARD_DOMAIN	DFTI_COMPLEX, or
	DFTI_REAL, or
	DFTI_CONJUGATE_EVEN (no default)
DFTI_FORWARD_SIGN	DFTI_NEGATIVE (default), or
	DFTI_POSITIVE
DFTI_PLACEMENT	DFTI_INPLACE (default), or
	DFTI_NOT_INPLACE
DFTI_COMPLEX_STORAGE	DFTI_COMPLEX_COMPLEX (default), or
	DFTI_COMPLEX REAL, or
	DFTI_REAL_REAL
DFTI_REAL_STORAGE	DFTI_REAL_REAL (default), or
	DFTI_REAL_COMPLEX
DFTI_CONJUGATE_EVEN_STORAGE	DFTI_COMPLEX_COMPLEX, or
	DFTI_COMPLEX_REAL (default), or
	DFTI_REAL_REAL (1-D transform only)
DFTI_PACKED_FORMAT	DFTI_CCS_FORMAT (default) or,
	DFTI_PACK_FORMAT or,
	DFTI_PERM_FORMAT

Table 11-7 lists the default values of the settable configuration parameters.

Table 11-7 Default Configuration Values of Settable Parameters

Named Constants	Default Value
DFTI_NUMBER_OF_TRANSFORMS	1
DFTI_NUMBER_OF_USER_THREADS	1
DFTI_FORWARD_SIGN	DFTI_NEGATIVE
DFTI_FORWARD_SCALE	1.0
DFTI_BACKWARD_SCALE	1.0
DFTI_PLACEMENT	DFTI_INPLACE
DFTI_COMPLEX_STORAGE	DFTI_COMPLEX_COMPLEX
DFTI_REAL_STORAGE	DFTI_REAL_REAL
DFTI_CONJUGATE_EVEN_STORAGE	DFTI_COMPLEX_REAL
DFTI_PACKED_FORMAT	DFTI_CCS_FORMAT
DFTI_DESCRIPTOR_NAME	no name, string of zero length
DFTI_INPUT_DISTANCE	0
DFTI_OUTPUT_DISTANCE	0
DFTI_INPUT_STRIDES	Tightly packed according to dimension, lengths, and storage
DFTI_OUTPUT_STRIDES	Same as above. See Strides for details
DFTI_INITIALIZATION_EFFORT	DFTI_MEDIUM
DFTI_ORDERING	DFTI_ORDERED
DFTI_WORKSPACE	DFTI_ALLOW
DFTI_TRANSPOSE	DFTI_NONE

Precision of transform

The configuration parameter `DFTI_PRECISION` denotes the floating-point precision in which the transform is to be carried out. A setting of `DFTI_SINGLE` stands for single precision, and a setting of `DFTI_DOUBLE` stands for double precision. The data is meant to be presented in this precision; the computation will be carried out in this precision; and the result will be delivered in this precision. This is one of the four settable configuration parameters that do not have default values. The user must set them explicitly, most conveniently at the call to descriptor creation function `DftiCreateDescriptor`.

Forward domain of transform

The general form of the discrete Fourier transform is

$$z_{k_1, k_2, \ldots, k_d} = \sigma \times \sum_{j_d=0}^{n_d-1} \ldots \sum_{j_2=0}^{n_2-1} \sum_{j_1=0}^{n_1-1} w_{j_1, j_2, \ldots, j_d} \exp\left(\delta i 2\pi \sum_{l=1}^{d} j_l k_l / n_l \right) \qquad (7.1)$$

for $k_l = 0, \pm1, \pm2, \ldots$, where σ is an arbitrary real-valued scale factor and $\delta = \pm1$. By default, the forward transform is defined by $\sigma = 1$ and $\delta = -1$. In most common situations, the domain of the forward transform, that is, the set where the input (periodic) sequence $\{w_{j_1, j_2, \ldots, j_d}\}$ belongs, can be either the set of complex-valued sequences, real-valued sequences, and complex-valued conjugate even sequences. The configuration parameter DFTI_FORWARD_DOMAIN indicates the domain for the forward transform. Note that this implicitly specifies the domain for the backward transform because of mathematical property of the DFT. See Table 11-8 for details.

Table 11-8 Correspondence of Forward and Backward Domain

Forward Domain		Implied Backward Domain
Complex	(DFTI_COMPLEX)	Complex
Real	(DFTI_REAL)	Conjugate Even
Conjugate Even	(DFTI_CONJUGATE_EVEN)	Real

On transforms in the real domain, some software packages only offer one "real-to-complex" transform. This in essence omits the conjugate even domain for the forward transform. The forward domain configuration parameter DFTI_FORWARD_DOMAIN is the second of four configuration parameters without default value.

Transform dimension and lengths

The dimension of the transform is a positive integer value represented in an integer scalar of type Integer. For one-dimensional transform, the transform length is specified by a positive integer value represented in an integer scalar of type Integer. For multi-dimensional (≥ 2) transform, the lengths of each of the dimension is supplied in an integer array. DFTI_DIMENSION and DFTI_LENGTHS are the remaining two of four configuration parameters without default.

As mentioned, these four configuration parameters do not have default value. They are most conveniently set at the descriptor creation function. For dimension and length configuration, they can only be set in the descriptor creation function, and not by the function DftiSetValue.

The other two configuration values can be changed through the function `DftiSetValue`, although this is not deemed common.

CAUTION. Changing the dimension and length would likely render the stride value inappropriate. Unless certain of otherwise, the user is advised to reconfigure the stride (see Strides).

Number of transforms

In some situations, the user may need to perform a number of DFT transforms of the same dimension and lengths. The most common situation would be to transform a number of one-dimensional data of the same length. This parameter has the default value of 1, and can be set to positive integer value by an `Integer` data type in Fortran and `long` data type in C. Data sets have no common elements. The distance parameter is obligatory if multiple number is more than one.

Sign and scale

The general form of the discrete Fourier transform is given by (7.1), for $k_l = 0, \pm1, \pm2, \ldots$, where σ is an arbitrary real-valued scale factor and $\delta = \pm1$. By default, the forward transform is defined by $\sigma = 1$ and $\delta = -1$, and the backward transform is defined by $\sigma = 1$ and $\delta = 1$. The user can change the definition of forward transform via setting the sign δ to be `DFTI_NEGATIVE` (default) or `DFTI_POSITIVE`. The sign of the backward transform is implicitly defined to be the negative of the sign for the forward transform.

The forward transform and backward transform are each associated with a scale factor σ of its own with default value of 1. The user can set one or both of them via the two configuration parameters `DFTI_FORWARD_SCALE` and `DFTI_BACKWARD_SCALE`. For example, for a one-dimensional transform of length *n*, one can use the default scale of 1 for the forward transform while setting the scale factor for backward transform to be 1/*n*, making the backward transform the inverse of the forward transform.

The scale factor configuration parameter should be set by a real floating-point data type of the same precision as the value for `DFTI_PRECISION`.

NOTE. The sign configuration is not supported. The forward transform is defined as $\delta = -1$.

Placement of result

By default, the computational functions overwrite the input data with the output result. That is, the default setting of the configuration parameter `DFTI_PLACEMENT` is `DFTI_INPLACE`. The user can change that by setting it to `DFTI_NOT_INPLACE`. Data sets have no common elements.

Packed formats

The result of the forward transform (i.e. in the frequency-domain) of real data is represented in several possible packed formats: ***Pack***, ***Perm***, or ***CCS***. The data can be packed due to the symmetry property of the DFT transform of a real data.

The ***CCS*** format stores the values of the first half of the output complex signal resulted from the forward DFT. Note that the signal stored in `CCS` format is one complex element longer. In `CCS` format, the output samples of the DFT are arranged as shown in Table 11-9 for one-dimensional DFT and in Table 11-10 for two-dimensional DFT.

The ***Pack*** format is a compact representation of a complex conjugate-symmetric sequence. The disadvantage of this format is that it is not the natural format used by the real DFT algorithms (“natural” in the sense that array is natural for complex DFTs). In `Pack` format, the output samples of the DFT are arranged as shown in Table 11-9 for one-dimensional DFT and in Table 11-11 for two-dimensional DFT.

The ***Perm*** format is an arbitrary permutation of the `Pack` format for even lengths and one is the same as the `Pack` format for odd lengths. In `Perm` format, the output samples of the DFT are arranged as shown in Table 11-9 for one-dimensional DFT and in Table 11-12 for two-dimensional DFT.

Table 11-9 Packed Format Output Samples

For (n = s*2)

DFT Real	0	1	2	3	...	n-2	n-1	n	n+1
CCS	R_0	0	R_1	I_1	...	$R_{n/2-1}$	$I_{n/2-1}$	$R_{n/2}$	0
Pack	R_0	R_1	I_1	R_2	...	$I_{n/2-1}$	$R_{n/2}$		
Perm	R_0	$R_{n/2}$	R_1	I_1	...	$R_{n/2-1}$	$I_{n/2-1}$		

For (n = s*2 + 1)

DFT Real	0	1	2	3	...	n-4	n-3	n-2	n-1	n	n+1
CCS	R_0	0	R_1	I_1	...	I_{s-2}	R_{s-1}	I_{s-1}	R_s	I_s	
Pack	R_0	R_1	I_1	R_2	...	R_{s-1}	I_{s-1}	R_{s-1}	I_s		
Perm	R_0	R_1	I_1	R_2	...	R_{s-1}	I_{s-1}	R_{s-1}	I_s		

Note that Table 11-9 uses the following notation for complex data entries:

$R_j = \text{Re } z_j$

$I_j = \text{Im } z_j$

See also Table 11-13 and Table 11-14.

Table 11-10 *CCS* Format Output Samples (Two-Dimensional Matrix (*m*+2)-by-(*n*+2))

For (*m* = s*2)								
z(1,1)	0	REz(1,2)	IMz(1,2)	...	REz(1,k)	IMz(1,k)	z(1,k+1)	0
0	0	0	0	...	0	0	0	0
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)	n/u	n/u
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)	n/u	n/u
...	...	...	...	...	...	...	n/u	n/u
REz(m/2,1)	REz(m/2,2)	REz(m/2,3)	REz(m/2,4)	...	REz(m/2,n-1)	REz(m/2,n)	n/u	n/u
IMz(m/2,1)	IMz(m/2,2)	IMz(m/2,3)	IMz(m/2,4)	...	IMz(m/2,n-1)	IMz(m/2,n)	n/u	n/u
z(m/2+1,1)	0	REz(m/2+1,2)	IMz(m/2+1,2)	...	REz(m/2+1,k)	IMz(m/2+1,k)	z(m/2+1,k+1)	0
0	0	0	0	...	0	0	n/u	n/u
For (*m* = s*2+1)								
z(1,1)	0	REz(1,2)	IMz(1,2)	...	REz(1,k)	IMz(1,k)	z(1,k+1)	0
0	0	0	0	...	0	0	0	0
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)	n/u	n/u
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)	n/u	n/u
...	...	...	...	...	...	...	n/u	n/u
REz(s,1)	REz(s,2)	REz(s,3)	REz(s,4)	...	REz(s,n-1)	REz(s,n)	n/u	n/u
IMz(s,1)	IMz(s,2)	IMz(s,3)	IMz(s,4)	...	IMz(s,n-1)	IMz(s,n)	n/u	n/u

* n/u - not used

Note that in the Table 11-10 (*n*+2) columns are used for even `n = k*2`, while *n* columns are used for odd `n = k*2+1`. In the latter case the first row is

```
z(1,1)    0   REz(1,2)  IMz(1,2)  …  REz(1,k)   IMz(1,k)
```

If *m* is even, the (*m*+1)-th row is

```
z(m/2+1,1)  0  REz(m/2+1,2) IMz(m/2+1,2) … REz(m/2+1,k) IMz(m/2+1,k)
```

Table 11-11 *Pack* Format Output Samples (Two-Dimensional Matrix *m*-by-*n*)

For (*m* = s*2)						
z(1,1)	REz(1,2)	IMz(1,2)	REz(1,3)	...	IMz(1,k)	z(1,k+1)
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)
...	...	...	...	...	...	...
REz(m/2,1)	REz(m/2,2)	REz(m/2,3)	REz(m/2,4)	...	REz(m/2,n-1)	REz(m/2,n)
IMz(m/2,1)	IMz(m/2,2)	IMz(m/2,3)	IMz(m/2,4)	...	IMz(m/2,n-1)	IMz(m/2,n)
z(m/2+1,1)	REz(m/2+1,2)	IMz(m/2+1,2)	REz(m/2+1,3)	...	IMz(m/2+1,k)	z(m/2+1,k+1)
For (*m* = s*2+1)						
z(1,1)	REz(1,2)	IMz(1,2)	REz(1,3)	...	IMz(1,k)	z(1,n/2+1)
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)
...	...	...	...	...	...	...
REz(s,1)	REz(s,2)	REz(s,3)	REz(s,4)	...	REz(s,n-1)	REz(s,n)
IMz(s,1)	IMz(s,2)	IMz(s,3)	IMz(s,4)	...	IMz(s,n-1)	IMz(s,n)

Table 11-12 *Perm* Format Output Samples (Two-Dimensional Matrix *m*-by-*n*)

For (*m* = s*2)						
z(1,1)	z(1,k+1)	REz(1,2)	IMz(1,2)	...	REz(1,k)	IMz(1,k)
z(m/2+1,1)	z(m/2+1,k+1)	REz(m/2+1,2)	IMz(m/2+1,2)	...	REz(m/2+1,k)	IMz(m/2+1,k)
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)
...	...	...	...	...	...	...
REz(m/2,1)	REz(m/2,2)	REz(m/2,3)	REz(m/2,4)	...	REz(m/2,n-1)	REz(m/2,n)
IMz(m/2,1)	IMz(m/2,2)	IMz(m/2,3)	IMz(m/2,4)	...	IMz(m/2,n-1)	IMz(m/2,n)
For (*m* = s*2+1)						
z(1,1)	z(1,k+1)	REz(1,2)	IMz(1,2)	...	REz(1,k)	IMz(1,k)
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)
...	...	...	...	...	...	...
REz(s,1)	REz(s,2)	REz(s,3)	REz(s,4)	...	REz(s,n-1)	REz(s,n)
IMz(s,1)	IMz(s,2)	IMz(s,3)	IMz(s,4)	...	IMz(s,n-1)	IMz(s,n)

Note that in the Table 11-11 and Table 11-12 for even number of columns `n = k*2`, while for odd number of columns `n = k*2+1` and the first row is

`z(1,1) REz(1,2) IMz(1,2) … REz(1,k) IMz(1,k)`

If `m` is even, the last row in `Pack` format and the second row in `Perm` format is

`z(m/2+1,1) REz(m/2+1,2) IMz(m/2+1,2) … REz(m/2+1,k) IMz(m/2+1,k)`

The tables for two-dimensional DFT use Fortran-interface conventions. For C-interface specifics in storing packed fata, see Storage schemes section below.
See also Table 11-15 and Table 11-16 for examples of Fortran-interface and C-interface formats.

Storage schemes

For each of the three domains `DFTI_COMPLEX`, `DFTI_REAL`, and `DFTI_CONJUGATE_EVEN` (for the forward as well as the backward operator), a subset of the four storage schemes `DFTI_COMPLEX_COMPLEX`, `DFTI_COMPLEX_REAL`, `DFTI_REAL_COMPLEX`, and `DFTI_REAL_REAL`. Specific examples are presented here to illustrate the storage schemes. See the document [3] for the rationale behind this definition of the storage schemes.

NOTE. The data is stored in the Fortran style only, that is, the real and imaginary parts are stored side by side.

Storage scheme for complex domain. This setting is recorded in the configuration parameter `DFTI_COMPLEX_STORAGE`. The three values that can be set are `DFTI_COMPLEX_COMPLEX`, `DFTI_COMPLEX_REAL`, and `DFTI_REAL_REAL`. Consider a one-dimensional *n*-length transform of the form

$$z_k = \sum_{j=0}^{n-1} w_j e^{-i2\pi jk/n}, \qquad w_j, z_k \in \mathbb{C}.$$

Assume the stride has default value (unit stride) and `DFTI_PLACEMENT` has the default in-place setting.

1. `DFTI_COMPLEX_COMPLEX` storage scheme (by default). A typical usage will be as follows.

```
COMPLEX :: X(0:n-1)
...some other code...
Status = DftiComputeForward( Desc_Handle, X )
```

On input,

X(j) $= w_j$, $j = 0,1,...,n-1$.

On output,

X(k) $= z_k$, $k = 0,1,...,n-1$.

2. DFTI_COMPLEX_REAL storage scheme. A typical usage will be as follows.

```
REAL :: X(0:2*n-1)
...some other code...
Status = DftiComputeForward( Desc_Handle, X )
```

On input,

X(2*j) $= \mathrm{Re}(w_j)$, X(2*j+1) $= \mathrm{Im}(w_j)$, $j = 0,1,...,n-1$.

On output,

X(2*k) $= \mathrm{Re}(z_k)$, X(2*k+1) $= \mathrm{Im}(z_k)$, $k = 0,1,...,n-1$.

The notations $\mathrm{Re}(w_j)$ and $\mathrm{Im}(w_j)$ are the real and imaginary parts of the complex number w_j.

3. DFTI_REAL_REAL storage scheme. A typical usage will be as follows.

```
REAL :: X(0:n-1), Y(0:n-1)
...some other code...
Status = DftiComputeForward( Desc_Handle, X, Y )
```

On input,

X(j) $= \mathrm{Re}(w_j)$, Y(j) $= \mathrm{Im}(w_j)$, $j = 0,1,...,n-1$.

On output,

X(k) $= \mathrm{Re}(z_k)$, Y(k) $= \mathrm{Im}(z_k)$, $k = 0,1,...,n-1$.

Storage scheme for the real and conjugate even domains. This setting for the storage schemes for these domains is recorded in the configuration parameters DFTI_REAL_STORAGE and DFTI_CONJUGATE_EVEN_STORAGE. Since a forward real domain corresponds to a conjugate even backward domain, they are considered together. The example uses one- and two-dimensional real to conjugate even transforms. In-place computation is assumed whenever possible (that is, when the input data type matches with the output data type).

Consider a one-dimensional n-length transform of the form

$$z_k = \sum_{j=0}^{n-1} w_j e^{-i2\pi jk/n}, \qquad w_j \in R, \; z_k \in C.$$

There is a symmetry:

For even n: `z(n/2+i) = conjg(z(n/2-i))`, $1 \le i \le n/2 - 1$, and moreover `z(0)` and `z(n/2)` are real values.

For odd n: `z(m+i) = conjg(z(m-i+1))`, $1 \le i \le m$, and moreover `z(0)` is real value.

m = floor(n/2).

Table 11-13 Comparison of the Storage Effects of Complex-to-Complex and Real-to-Complex DFTs for Forward Transform

N=8

Input Vectors			Output Vectors				
Complex DFT		**Real DFT**	**complex DFT**		**real DFT**		
Complex Data		**Real Data**	**Complex Data**		**Real Data**		
Real	**Imaginary**		**Real**	**Imaginary**	**CCS**	**Pack**	**Perm**
w0	0.000000	w0	z0	0.000000	z0	z0	z0
w1	0.000000	w1	Re(z1)	Im(z1)	0.000000	Re(z1)	z4
w2	0.000000	w2	Re(z2)	Im(z2)	Re(z1)	Im(z1)	Re(z1)
w3	0.000000	w3	Re(z3)	Im(z3)	Im(z1)	Re(z2)	Im(z1)
w4	0.000000	w4	z4	0.000000	Re(z2)	Im(z2)	Re(z2)
w5	0.000000	w5	Re(z3)	-Im(z3)	Im(z2)	Re(z3)	Im(z2)
w6	0.000000	w6	Re(z2)	-Im(z2)	Re(z3)	Im(z3)	Re(z3)
w7	0.000000	w7	Re(z1)	-Im(z1)	Im(z3)	z4	Im(z3)
					z4		
					0.000000		

N=7

Input Vectors			Output Vectors				
`Complex DFT`		`Real DFT`	`complex DFT`		`real DFT`		
Complex Data		Real Data	Complex Data		Real Data		
Real	**Imaginary**		**Real**	**Imaginary**	**CCS**	**Pack**	**Perm**
w0	0.000000	w0	z0	0.000000	z0	z0	z0
w1	0.000000	w1	Re(z1)	Im(z1)	0.000000	Re(z1)	Re(z1)
w2	0.000000	w2	Re(z2)	Im(z2)	Re(z1)	Im(z1)	Im(z1)
w3	0.000000	w3	Re(z3)	Im(z3)	Im(z1)	Re(z2)	Re(z2)
w4	0.000000	w4	Re(z3)	-Im(z3)	Re(z2)	Im(z2)	Im(z2)
w5	0.000000	w5	Re(z2)	-Im(z2)	Im(z2)	Re(z3)	Re(z3)
w6	0.000000	w6	Re(z1)	-Im(z1)	Re(z3)	Im(z3)	Im(z3)
					Im(z3)		

Table 11-14 Comparison of the Storage Effects of Complex-to-Complex and Complex-to-Real DFTs for Backward Transform

N=8

Output Vectors			Input Vectors				
Complex DFT		**Real DFT**	**complex DFT**				
Complex Data		**Real Data**	**Complex Data**				
Real	**Imaginary**		**Real**	**Imaginary**	**CCS**	**Pack**	**Perm**
w0	0.000000	w0	z0	0.000000	z0	z0	z0
w1	0.000000	w1	Re(z1)	Im(z1)	0.000000	Re(z1)	z4
w2	0.000000	w2	Re(z2)	Im(z2)	Re(z1)	Im(z1)	Re(z1)
w3	0.000000	w3	Re(z3)	Im(z3)	Im(z1)	Re(z2)	Im(z1)
w4	0.000000	w4	z4		Re(z2)	Im(z2)	Re(z2)
w5	0.000000	w5	Re(z3)	-Im(z3)	Im(z2)	Re(z3)	Im(z2)
w6	0.000000	w6	Re(z2)	-Im(z2)	Re(z3)	Im(z3)	Re(z3)
w7	0.00000	w7	Re(z1)	-Im(z1)	Im(z3)	z4	Im(z3)
					z4		
					0.000000		

N=7

Output Vectors			Input Vectors				
Complex DFT		**Real DFT**	**complex DFT**		**real DFT**		
Complex Data		**Real Data**	**Complex Data**		**Real Data**		
Real	**Imaginary**		**Real**	**Imaginary**	**CCS**	**Pack**	**Perm**
w0	0.000000	w0	z0	0.000000	z0	z0	z0
w1	0.000000	w1	Re(z1)	Im(z1)	0.000000	Re(z1)	Re(z1)
w2	0.000000	w2	Re(z2)	Im(z2)	Re(z1)	Im(z1)	Im(z1)
w3	0.000000	w3	Re(z3)	Im(z3)	Im(z1)	Re(z2)	Re(z2)
w4	0.000000	w4	Re(z3)	-Im(z3)	Re(z2)	Im(z2)	Im(z2)
w5	0.000000	w5	Re(z2)	-Im(z2)	Im(z2)	Re(z3)	Re(z3)
w6	0.000000	w6	Re(z1)	-Im(z1)	Re(z3)	Im(z3)	Im(z3)
					Im(z3)		

Assume that the stride has the default value (unit stride).

This complex conjugate-symmetric vector can be stored in the complex array of size `m+1` or in the real array of size `2m+2` or `2m` depending on packed format.

Each of the real-to-complex routines computes the forward DFT of a two-dimensional real matrix according to the mathematical equation

$$z_{i,j} = \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} t_{k,l} * w_m^{-i*k} * w_n^{-j*l}, \quad 0 \le i \le m-1, \; 0 \le j \le n-1$$

$t_{k,l} = \mathrm{cmplx}(r_{k,l}, 0)$, where $r_{k,l}$ is a real input matrix, $0 \le k \le m-1$, $0 \le l \le n-1$. The mathematical result $z_{i,j}$, $0 \le i \le m-1$, $0 \le j \le n-1$, is the complex matrix of size (m,n). Each column is the complex conjugate-symmetric vector as follows:

For even m:

for $0 \le j \le n-1$,

$z(m/2+i, j) = \mathrm{conjg}(z(m/2-i, j))$, $1 \le i \le m/2-1$.

Moreover, $z(0,j)$ and $z(m/2,j)$ are real values for $j=0$ and $j=n/2$.

For odd m:

for $0 \le j \le n-1$,

$z(s+i, j) = \mathrm{conjg}(z(s-i, j))$, $1 \le i \le s-1$,
where $s = \mathrm{floor}(m/2)$.

Moreover, $z(0,j)$ are real values for $j=0$ and $j=n/2$.

This mathematical result can be stored in the real two-dimensional array of size $(m+2, n+2)$ or (m,n), or in the complex two-dimensional array of size $(m/2+1, n+1)$ for Fortran-interface and in the complex two-dimensional array of size $(m+1, n/2+1)$ for C-interface.

Since the multidimensional array data are arranged differently in Fortran and C (see Strides), the output array that holds the computational result contains complex conjugate-symmetric columns (for Fortran) or complex conjugate-symmetric rows (for C).

The following tables give examples of output data layout in `Pack` format for a forward two-dimensional real-to-complex DFT of a 6-by-4 real matrix. Note that the same layout is used for the input data of the corresponding backward complex-to-real DFT.

Table 11-15 Fortran-interface Data Layout for a 6-by-4 Matrix

z(1,1)	Re z(1,2)	Im z(1,2)	z(1,3)
Re z(2,1)	Re z(2,2)	Re z(2,3)	Re z(2,4)
Im z(2,1)	Im z(2,2)	Im z(2,3)	Im z(2,4)
Re z(3,1)	Re z(3,2)	Re z(3,3)	Re z(3,4)
Im z(3,1)	Im z(3,2)	Im z(3,3)	Im z(3,4)
z(4,1)	Re z(4,2)	Im z(4,2)	z(4,3)

For the above example, the stride array is taken to be (0, 1, 6).

Table 11-16 C-interface Data Layout for a 6-by-4 Matrix

z(1,1)	Re z(1,2)	Im z(1,2)	z(1,3)
Re z(2,1)	Re z(2,2)	Im z(2,2)	Re z(2,3)
Im z(2,1)	Re z(3,2)	Im z(3,2)	Im z(2,3)
Re z(3,1)	Re z(4,2)	Im z(4,2)	Re z(3,3)
Im z(3,1)	Re z(5,2)	Im z(5,2)	Im z(3,3)
z(4,1)	Re z(6,2)	Im z(6,2)	z(4,3)

For the second example, the stride array is taken to be /0, 4, 1/.
See also Packed formats.

1. `DFTI_REAL_REAL` for real domain, `DFTI_COMPLEX_REAL` for conjugate even domain (by default). A typical usage will be as follows.

```
// m = floor( n/2 )
REAL :: X(0:2*m+1)
...some other code...
...assuming inplace...
Status = DftiComputeForward( Desc_Handle, X )
```

On input,

$X(j) = w_j ,\ j = 0,1,...,n-1$.

On output,

Output data stored in one of formats: Pack, Perm or CCS (see “Packed formats”).

CCS format: X(2*k) = Re(z_k) , X(2*k+1) = Im(z_k) , k = 0,1,...,m.

Pack format: even n: X(0) = Re(z_0), X(2*k-1) = Re(z_k), X(2*k) = Im(z_k), k = 1,...,m-1, and X(n-1) = Re(z_m)

odd n: X(0) = Re(z_0), X(2*k-1) = Re(z_k), X(2*k) = Im(z_k), k = 1,...,m

Perm format: even n: X(0) = Re(z_0), X(1) = Re(z_m), X(2*k) = Re(z_k) , X(2*k+1) = Im(z_k) , k = 1,...,m-1,

odd n: X(0) = Re(z_0), X(2*k-1) = Re(z_k), X(2*k) = Im(z_k), k = 1,...,m.

2. DFTI_REAL_REAL for real domain, DFTI_COMPLEX_REAL for conjugate even domain (by default). A typical usage will be as follows.

```
// m = floor( n/2 )
REAL :: X(0:n-1)
REAL :: Y(0:2*m+1)
...some other code...
...assuming out-of-place...
Status = DftiComputeForward( Desc_Handle, X, Y )
```

On input,

X(j) = w_j , j = 0,1,...,n-1 .

On output,

Output data stored in one of formats: Pack, Perm or CCS (see “Packed formats”).

CCS format: Y(2*k) = Re(z_k) , Y(2*k+1) = Im(zk) , k = 0,1,...,m.

Pack format: even n: Y(0) = Re(z_0), Y(2*k-1) = Re(z_k), Y(2*k) = Im(z_k), k = 1,...,m-1, and Y(n-1) = Re(z_m)

odd n: Y(0) = Re(z_0), Y(2*k-1) = Re(z_k), Y(2*k) = Im(z_k), k = 1,...,m

Perm format: even n: Y(0) = Re(z_0), Y(1) = Re(z_m), Y(2*k) = Re(z_k) , Y(2*k+1) = Im(z_k) , k = 1,...,m-1,

odd n: Y(0) = Re(z_0), Y(2*k-1) = Re(z_k), Y(2*k) = Im(z_k), k = 1,...,m.

Notice that if the stride of the output array is not set to the default value unit stride, the real and imaginary parts of one complex element will be placed with this stride.

For example:

CCS format: $Y(2*k*s) = Re(z_k)$, $Y(2*k+1*s) = Im(zk)$, k = 0,1,...,m, s - stride.

3. `DFTI_REAL_REAL` for real domain, `DFTI_COMPLEX_COMPLEX` for conjugate even domain. A typical usage will be as follows.

```
// m = floor( n/2 )
REAL :: X(0:n-1)
COMPLEX :: Y(0:m)
...some other code...
...out of place transform...
Status = DftiComputeForward( Desc_Handle, X, Y )
```

On input,

$X(j) = w_j$, j = 0,1,...,n-1 .

On output,

$Y(k) = z_k$, k = 0,1,...,m .

4. `DFTI_REAL_REAL` for real domain, `DFTI_REAL_REAL` for conjugate even domain. This storage scheme for conjugate even domain is applicable for one-dimensional transform only. A typical usage will be as follows.

```
// m = floor( n/2 )
REAL :: X(0:n-1)
...some other code...
...assuming inplace...
Status = DftiComputeForward( Desc_Handle, X )
```

On input,

$X(j) = w_j$, j = 0,1,...,n-1 .

On output,

$X(k) = Re(z_k)$, k = 0,1,...,m .

and

$X(n-k) = Im(z_k)$, k = 1,2,...,m-1 .

5. `DFTI_REAL_COMPLEX` for real domain, `DFTI_COMPLEX_COMPLEX` for conjugate even domain. A typical usage will be as follows.

```
// m = floor( n/2 )
COMPLEX :: X(0:n-1)
...some other code...
...inplace transform...
Status = DftiComputeForward( Desc_Handle, X )
```

On input,

$X(j) = w_j$, $j = 0,1,...,n-1$.

That is, the imaginary parts of X(*j*) are zero. On output,

$Y(k) = z_k$, $k = 0,1,...,m$.

where m is $\lfloor n/2 \rfloor$.

6. `DFTI_REAL_COMPLEX` for real domain, `DFTI_COMPLEX_REAL` for conjugate even domain. A typical usage will be as follows.

```
// m = floor( n/2 )
COMPLEX :: X(0:n-1)
REAL :: Y(0:2*m+1)
...some other code...
...not inplace...
Status = DftiComputeForward( Desc_Handle, X, Y )
```

On input,

$X(j) = w_j$, $j = 0,1,...,n-1$.

On output,

Output data stored in one of formats: Pack, Perm or CCS (see “Packed formats”).

CCS format: $Y(2*k) = Re(z_k)$, $Y(2*k+1) = Im(z_k)$, $k = 0,1,...,m$.

Pack format: even n: $Y(0) = Re(z_0)$, $Y(2*k-1) = Re(z_k)$, $Y(2*k) = Im(z_k)$,
$k = 1,...,m-1$, and $Y(n-1) = Re(z_m)$

odd n: $Y(0) = Re(z_0)$, $Y(2*k-1) = Re(z_k)$, $Y(2*k) = Im(z_k)$, $k = 1,...,m$

Perm format: even n: Y(0) = Re(z_0), Y(1) = Re(z_m), Y(2*k) = Re(z_k) , Y(2*k+1) = Im(z_k) , k = 1,...,m-1,

odd n: Y(0) = Re(z_0), Y(2*k-1) = Re(z_k), Y(2*k) = Im(z_k), k = 1,...,m.

6. `DFTI_REAL_COMPLEX` for real domain, `DFTI_REAL_REAL` for conjugate even domain. This storage scheme for conjugate even domain is applicable for one-dimensional transform only. A typical usage will be as follows.

```
// m = floor( n/2 )
COMPLEX :: X(0:n-1)
REAL :: Y(0:n-1)
...some other code...
...not inplace...
Status = DftiComputeForward( Desc_Handle, X, Y )
```

On input,

X(*j*) = w_j , *j* = 0,1,...,n-1 .

On output,

Y(*k*) = Re(z_k) , *k* = 0,1,...,m .

and

Y(*n*-*k*) = Im(z_k) , *k* = 1,2,...,m-1 .

Number of user threads

Customer application can be parallelized by using the following techniques:

1. You do not create threads in your application but specify the parallel mode within the DFT module of Intel MKL. See *Intel MKL Technical User Notes* document for more information on how to do this.
2. You create threads in application yourself and have each thread perform all stages of DFT implementation including descriptor initialization, DFT computation, and descriptor deallocation. In this case each descriptor is used only within its corresponding thread.
3. You create threads after initializing the DFT descriptor. This implies that threading is employed for parallel DFT computation only, and the descriptor is freed after return from the parallel region. In this case each thread uses the same descriptor.

For the first and second cases listed above, set the parameter DFTI_NUMBER_OF_USER_THREADS to 1 (its default value), since each particular descriptor instance is used only in a single thread.

In case 3, you must use the DftiSetValue() function to set the DFTI_NUMBER_OF_USER_THREADS to the actual number of DFT computation threads, because multiple threads will be using the same descriptor. If this setting is not done, your program will work incorrectly or fail, since the descriptor contains individual data for each thread.

WARNING.
1. It is not recommended to simultaneously parallelize your program and employ the Intel MKL internal threading because this will slow down performance. Note that in case 3 above, DFT computation is automatically initiated in a single threading mode.
2. The number of threads must not be changed after DFT initialization by the DftiCommitDescriptor() function is done. For example, do not use the OMP function omp_set_max_threads() for this purpose.

See Example C-21, Example C-22, and Example C-23 in Appendix C.

Input and output distances

DFT interface in Intel MKL allows the computation of multiple number of transforms. Consequently, the user needs to be able to specify the data distribution of these multiple sets of data. This is accomplished by the distance between the first data element of the consecutive data sets. This parameter is obligatory if multiple number is more than one. Data sets don't have any common elements.The following example illustrates the specification. Consider computing the forward DFT on three 32-length complex sequences stored in X(0:31, 1), X(0:31, 2), and X(0:31, 3). Suppose the results are to be stored in the locations Y(0:31, k), k = 1, 2, 3, of the array Y(0:63, 3). Thus the input distance is 32, while the output distance is 64. Notice that the data and result parameters in computation functions are all declared as assumed-size rank-1 array DIMENSION(0:*). Therefore two-dimensional array must be transformed to one-dimensional array by EQUIVALENCE statement or other facilities of Fortran. Here is the code fragment:

```
Complex ::  X_2D(0:31,3), Y_2D(0:63, 3)
Complex ::  X(96), Y(192)
Equivalence (X_2D, X)
Equivalence (Y_2D, Y)
...................
```

```
Status = DftiCreateDescriptor(Desc_Handle, DFTI_SINGLE,
                              DFTI_COMPLEX, 1, 32)
Status = DftiSetValue(Desc_Handle, DFTI_NUMBER_OF_TRANSFORMS, 3)
Status = DftiSetValue(Desc_Handle, DFTI_INPUT_DISTANCE, 32)
Status = DftiSetValue(Desc_Handle, DFTI_OUTPUT_DISTANCE, 64)
Status = DftiSetValue(Desc_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE)
Status = DftiCommitDescriptor(Desc_Handle)
Status = DftiComputeForward(Desc_Handle, X, Y)
Status = DftiFreeDescriptor(Desc_Handle)
```

Strides

In addition to supporting transforms of multiple number of datasets, DFT interface supports non-unit stride distribution of data within each data set. Consider the following situation where a 32-length DFT is to be computed on the sequence x_j, $0 \le j < 32$. The actual location of these values are in `X(5)`, `X(7)`, ..., `X(67)` of an array `X(1:68)`. The stride accommodated by DFT interface consists of a displacement from the first element of the data array L_0, (4 in this case), and a constant distance of consecutive elements L_1 (2 in this case). Thus for the Fortran array `X`

$x_j = \mathrm{X}(1 + L_0 + L_1 * j) = \mathrm{X}(5 + L_1 * j)$.

This stride vector (4,2) is provided by a length-2 rank-1 integer array:

```
COMPLEX :: X(68)
INTEGER :: Stride(2)
...................
Status = DftiCreateDescriptor(Desc_Handle, DFTI_SINGLE,
                              DFTI_COMPLEX, 1, 32)
Stride = (/ 4, 2 /)
Status = DftiSetValue(Desc_Handle, DFTI_INPUT_STRIDES, Stride)
Status = DftiSetValue(Desc_Handle, DFTI_OUTPUT_STRIDES, Stride)
Status = DftiCommitDescriptor(Desc_Handle)
Status = DftiComputeForward(Desc_Handle, X)
Status = DftiFreeDescriptor(Desc_Handle)
```

In general, for a d-dimensional transform, the stride is provided by a $d+1$-length integer vector $(L_0, L_1, L_2, ..., L_d)$ with the meaning:

L_0 = displacement from the first array element

L_1 = distance between consecutive data elements in the first dimension

L_2 = distance between consecutive data elements in the second dimension

... = ...

L_d = distance between consecutive data elements in the *d*-th dimension.

A *d*-dimensional data sequence

$$x_{j_1, j_2, \ldots, j_d}\ , \qquad 0 \le j_i < J_i\,, \qquad 1 \le i \le d$$

will be stored in the rank-1 array `X` by the mapping

$$x_{j_1, j_2, \ldots, j_d} = \mathrm{X}(\text{first index} + L_0 + j_1 L_1 + j_2 L_2 + \ldots + j_d L_d)\ .$$

For multiple transforms, the value L_0 applies to the first data sequence, and L_j , j = 1, 2,..., d apply to all the data sequences.

In the case of a single one-dimensional sequence, L_1 is simply the usual stride. The default setting of strides in the general multi-dimensional situation corresponds to the case where the sequences are distributed tightly into the array:

$$L_1 = 1,\ L_2 = J_1,\ L_3 = J_1 J_2\ ,\ldots,\ L_d = \prod_{i=1}^{d-1} J_i$$

Both the input data and output data have a stride associated with it. The default is set in accordance with the data to be stored contiguously in memory in a way that is natural to the language.
See Example C-20 as an illustration on how to use the configuration parameters discussed above.

Initialization Effort

In modern approaches to constructing fast algorithms (FFT) for DFT computations, one often has a flexibility of spending more effort in initializing (preparing for) an FFT algorithm to buy higher efficiency in the computation on actual data to follow. Advanced DFT functions in Intel MKL accommodate this situation through the configuration parameter `DFTI_INITIALIZATION_EFFORT`. The three configuration values are `DFTI_LOW`, `DFTI_MEDIUM` (default), and `DFTI_HIGH`. Note that specific implementations of DFT interface may or may not make use of this setting (see *MKL Release Notes* for implementation details).

Ordering

It is well known that a number of FFT algorithms apply an explicit permutation stage that is time consuming [4]. The exclusion of this step is similar to applying DFT to input data whose order is scrambled, or allowing a scrambled order of the DFT results. In applications such as convolution and power spectrum calculation, the order of result or data is unimportant and thus permission of scrambled order is attractive if it leads to higher performance. Three following options are available in Intel MKL:

1. `DFTI_ORDERED`: Forward transform data ordered, backward transform data ordered (default option).
2. `DFTI_BACKWARD_SCRAMBLED`: Forward transform data ordered, backward transform data scrambled.
3. `DFTI_FORWARD_SCRAMBLED`: Forward transform data scrambled, backward transform data ordered.

Table 11-17 tabulates the effect on this configuration setting.

Table 11-17 Scrambled Order Transform

	DftiComputeForward	**DftiComputeBackward**
DFTI_ORDERING	**Input → Output**	**Input → Output**
DFTI_ORDERED	ordered → ordered	ordered → ordered
DFTI_BACKWARD_SCRAMBLED	ordered → scrambled	scrambled → ordered
DFTI_FORWARD_SCRAMBLED	scrambled → ordered	ordered → scrambled

Note that meaning of the latter two options are "allow scrambled order if practical." There are situations where in fact allowing out of order data gives no performance advantage, and thus an implementation may choose to ignore the suggestion. Strictly speaking, the normal order is also a scrambled order, the trivial one.

When the ordering setting is other than the default `DFTI_ORDERED`, the user may need to know the actual ordering of the input and output data. The ordering of the data in the forward domain is obtained through reading (getting) the configuration parameter `DFTI_FORWARD_ORDERING`; and the ordering of the data in the reverse domain is obtained through reading (getting) the configuration parameter `DFTI_BACKWARD_ORDERING`. The configuration values are integer vectors, thus provided by pointer to any integer array. We now describe how these integer values specify the actual scrambling of data.

All scramblings involved are digit reversal along one single dimension. Precisely, a length J is factored into K ordered factors $D_1, D_2, ..., D_K$. Any index i, $0 \le i < n$, can be expressed uniquely as K digits $i_1, i_2, ..., i_K$ where
$0 \le i_l < D_l$ and

$i = i_1 + i_2 D_1 + i_3 D_1 D_2 + ... + i_K D_1 D_2 ... D_{K-1}$.

A digit reversal permutation scram(i) is given by

scram(i) = $i_K + i_{K-1} D_K + i_{K-2} D_K D_{K-1} + ... + i_1 D_K D_{K-1} ... D_2$

Factoring J into one factor J leads to no scrambling at all, that is, scram(i) = i. Note that the factoring does not need to correspond exactly to the number of "butterfly" stages to be carried out. In fact, the computation routine in its initialization stage determines if a scrambled order in some or all of the dimensions can result in performance gain. The digits of the digit reversal are recorded and stored in the descriptor. These digits can be obtained by calling a corresponding inquiry routine that returns a pointer to an integer array. The first element is $K^{(1)}$, which is the number of digits for the first dimension, followed by $K^{(1)}$ values of the corresponding digits. If the dimension is higher than one, the next integer value is $K^{(2)}$, etc.

Simple permutation such as mod-p sort [4] is a special case of digit reversal. Hence this option could be useful for high-performance implementation of one-dimensional DFT via a "six-step" or "four-step" framework [4].

The user can check the scrambling setting on the forward data and reverse data. This information is returned as an integer vector containing a number of sequence ($K, D_1, D_2, ..., D_K$), one for each dimension. Thus the first element indicates how many D's will follow. The inquiry routine allocates memory, fills it will this information, and returns a pointer to the memory location.

Workspace

Some FFT algorithms do not require a scratch space for permutation purposes. The user can choose between the setting of `DFTI_ALLOW` (default) and `DFTI_AVOID` for the option `DFTI_WORKSPACE`. Note that the setting `DFTI_AVOID` is meant to be "avoid if practical," hence allowing the implementation the flexibility to use workspace regardless of the setting.

Transposition

This is an option that allows for the result of a high-dimensional transform to be presented in a transposed manner. The default setting is `DFTI_NONE` and can be set to `DFTI_ALLOW`. Similar to that of scrambled order, sometimes in higher dimension transform, performance can be gained if the result is delivered in a transposed manner. DFT interface offers an option for the output be returned in a transposed form if performance gain is possible. Since the generic stride specification is naturally suited for representation of transposition, this option allows the strides for the output to be possibly different from those originally specified by the user. Consider an example where a two-dimensional result

$$y_{j_1, j_2}, \quad 0 \le j_i < n_i,$$

is expected. Originally the user specified that the result be distributed in the (flat) array `Y` in with generic strides $L_1 = 1$ and $L_2 = n_1$. With the transposition option, the computation may actually return the result into `Y` with stride $L_1 = n_2$ and $L_2 = 1$. These strides can be obtained from an appropriate inquiry function. Note also that in dimension 3 and above, transposition means an arbitrary permutation of the dimension.

Fast Fourier Transforms 12

This chapter describes the one- and two-dimensional fast Fourier transform (FFT) routines implemented in Intel® MKL. The FFT routines work with transforms of a power of 2 length and are supported to provide compatibility with previous versions of the library.

For a more general set of Discrete Fourier Transform functions in Intel MKL, refer to Discrete Fourier Transform Functions in this manual.

Although Intel MKL still supports the FFT interface described later in this chapter, users are encouraged to migrate to the newer DFT functions in their application programs. Unlike the FFT routines, the DFT routines support transforms of up to the dimension of seven, and transform lengths of other than powers of 2 mixed radix.

This chapter contains the following major sections:

- One-dimensional FFTs
- Two-dimensional FFTs

Each of the major sections contains the description of three groups of the FFTs.

One-dimensional FFTs

The one-dimensional FFTs include the following groups:

- Complex-to-Complex Transforms
- Real-to-Complex Transforms
- Complex-to-Real Transforms.

All one-dimensional FFTs are in-place. The transform length must be a power of 2. The complex-to-complex transform routines perform both forward and inverse transforms of a complex vector. The real-to-complex transform routines perform forward transforms of a real vector. The complex-to-real transform routines perform inverse transforms of a complex conjugate-symmetric vector, which is packed in a real array.

Data Storage Types

Each FFT group contains two sets of FFTs having the similar functionality: one set is used for the Fortran-interface and the other for the C-interface. The former set stores the complex data as a Fortran complex data type, while the latter stores the complex data as float arrays of real and imaginary parts separately. These sets are distinguished by naming the FFTs within each set. The names of the FFTs used for the C-interface have the letter "c" added to the end of the FFTs' Fortran names. For example, the names of the `cfft1d/zfft1d` FFTs for the corresponding C-interface routines are `cfft1dc/zfft1dc`. All names of the C-type data items are lower case.

Table 12-1 lists the one-dimensional FFT routine groups and the data types associated with them.

Table 12-1 One-dimensional FFTs: Names and Data Types

Group	Stored as Fortran Complex Data	Stored as C Real Data	Data Types	Description
Complex-to - Complex	`cfft1d/ zfft1d`	`cfftldc/ zfftldc`	c, z	Transform complex data to complex data.
Real-to-Complex	`scfft1d/ dzfft1d`	`scfft1dc/ dzfft1dc`	sc, dz	Transform forward real-to-complex data. Complement `csfft1d/zdfft1d` and `csfft1dc/zdfft1dc` FFTs.
Complex-to-Real	`csfft1d/ zdfft1d`	`csfft1dc/ zdfft1dc`	cs, zd	Transform inverse complex-to-real data. Complement `scfft1d/dzfft1d` and `scfft1dc/dzfft1dc` FFTs.

Data Structure Requirements

For C-interface, storage of the complex-to-complex transform routines data requires separate float arrays for the real and imaginary parts. The real-to-complex and complex-to-real pairs require a single float input/output array.

The C-interface requires scalar values to be passed by value.

All transforms require additional memory to store the transform coefficients. When performing multiple FFTs of the same dimension, the table of coefficients should be created only once and then used on all the FFTs afterwards. Using the same table rather than creating it repeatedly for each FFT produces an obvious performance gain.

Complex-to-Complex One-dimensional FFTs

Each of the complex-to-complex routines computes a forward or inverse FFT of a complex vector. The forward FFT is computed according to the mathematical equation

$$z_j = \sum_{k=0}^{n-1} r_k * w^{-j*k}, \quad 0 \le j \le n-1$$

The inverse FFT is computed according to the mathematical equation

$$r_j = \frac{1}{n} \sum_{k=0}^{n-1} z_k * w^{j*k}, \quad 0 \le j \le n-1$$

where $w = \exp\left[\frac{2\pi i}{n}\right]$, *i* being the imaginary unit.

The operation performed by the complex-to-complex routines is determined by the value of the *isign* parameter used by each of these routines.

If *isign* = -1, perform the forward FFT where input and output are in normal order.

If *isign* = +1, perform the inverse FFT where input and output are in normal order.

If *isign* = -2, perform the forward FFT where input is in normal order and output is in bit-reversed order.

If *isign* = +2, perform the inverse FFT where input is in bit-reversed order and output is in normal order.

If *isign* = 0, initialize FFT coefficients for both the forward and inverse FFTs.

The above equations apply to all FFTs with all data types indicated in Table 12-1.

To compute a forward or inverse FFT of a given length, first initialize the coefficients by calling the function with *isign* = 0. Thereafter, any number of transforms of the same length can be computed by calling the function with *isign* = +1, -1, +2, -2.

cfft1d/zfft1d

Fortran-interface routines. Compute the forward or inverse FFT of a complex vector (in-place)

Syntax

```
call cfft1d ( r, n, isign, wsave )
call zfft1d ( r, n, isign, wsave )
```

Description

The operation performed by the `cfft1d/zfft1d` routines is determined by the value of *`isign`*. See the equations of the operations for the Complex-to-Complex One-dimensional FFTs above.

Input Parameters

`r`	`COMPLEX` for `cfft1d` `DOUBLE COMPLEX` for `zfft1d` Array, `DIMENSION` at least `(`*`n`*`)`. Contains the complex vector on which the transform is to be performed. Not referenced if *`isign`* = `0`.
`n`	`INTEGER`. Transform length; *`n`* must be a power of 2.
`isign`	`INTEGER`. Flag indicating the type of operation to be performed: if *`isign`* = `0`, initialize the coefficients *`wsave`*; if *`isign`* = `-1`, perform the forward FFT where input and output are in normal order; if *`isign`* = `+1`, perform the inverse FFT where input and output are in normal order; if *`isign`* = `-2`, perform the forward FFT where input is in normal order and output is in bit-reversed order; if *`isign`* = `+2`, perform the inverse FFT where input is in bit-reversed order and output is in normal order.
`wsave`	`COMPLEX` for `cfft1d` `DOUBLE COMPLEX` for `zfft1d` Array, `DIMENSION` at least `((3*`*`n`*`)/2)`. If *`isign`* = `0`, then *`wsave`* is an output parameter. Otherwise, *`wsave`* contains the FFT coefficients initialized on a previous call with *`isign`* = `0`.

Output Parameters

r	Contains the complex result of the transform depending on *isign*. Does not change if *isign* = 0.
wsave	If *isign* = 0, *wsave* contains the initialized FFT coefficients. Otherwise, *wsave* does not change.

cfft1dc/zfft1dc

C-interface routines. Compute the forward or inverse FFT of a complex vector (in-place).

Syntax

```
void cfft1dc (float* r, float* i, int n, int isign, float* wsave)
void zfft1dc (double* r, double* i, int n, int isign, double* wsave)
```

Description

The operation performed by the `cfft1dc/zfft1dc` routines is determined by the value of *isign*. See the equations of the operations for the Complex-to-Complex One-dimensional FFTs.

Input Parameters

r
: `float*` for `cfft1dc`
`double*` for `zfft1dc`
Pointer to an array of size at least (*n*). Contains the real parts of complex vector to be transformed. Not referenced if *isign* = 0.

i
: `float*` for `cfft1dc`
`double*` for `zfft1dc`

  Pointer to an array of size at least (*n*). Contains the imaginary parts of complex vector to be transformed.

  Not referenced if *isign* = 0.

n
: `int`. Transform length; *n* must be a power of 2.

isign
: `int`. Flag indicating the type of operation to be performed:
if *isign* = 0, initialize the coefficients *wsave*;
if *isign* = -1, perform the forward FFT where input and output are in normal

order;
if *isign* = +1, perform the inverse FFT where input and output are in normal order;
if *isign* = -2, perform the forward FFT where input is in normal order and output is in bit-reversed order;
if *isign* = +2, perform the inverse FFT where input is in bit-reversed order and output is in normal order.

wsave — `float*` for `cfft1dc`
`double*` for `zfft1dc`
Pointer to an array of size at least `(3*n)`. If *isign* = 0, then *wsave* is an output parameter. Otherwise, *wsave* contains the FFT coefficients initialized on a previous call with *isign* = 0.

Output Parameters

r — Contains the real part of the transform depending on *isign*. Does not change if *isign* = 0.

i — Contains the imaginary part of the transform depending on *isign*.. Does not change if *isign* = 0.

wsave — If *isign* = 0, *wsave* contains the initialized FFT coefficients. Otherwise, *wsave* does not change.

Real-to-Complex One-dimensional FFTs

Each of the real-to-complex routines computes forward FFT of a real input vector according to the mathematical equation

$$z_j = \sum_{k=0}^{n-1} t_k * w^{-j*k}, \quad 0 \le j \le n-1$$

for t_k = `cmplx(`r_k`,0)`, where r_k is the real input vector, $0 \le k \le n-1$.
The mathematical result z_j, $0 \le j \le n-1$, is the complex conjugate-symmetric vector, where `z(n/2+i) = conjg(z(n/2-i))`, $1 \le i \le n/2-1$, and moreover `z(0)` and `z(n/2)` are real values.

This complex conjugate-symmetric (CCS) vector can be stored in the complex array of size `(n/2+1)` or in the real array of size `(n+2)`. The data storage of the CCS format is defined later for Fortran-interface and C-interface routines separately.

Table 12-2 shows a comparison of the effects of performing the `cfft1d/ zfft1d` complex-to-complex FFT on a vector of length `n=8` in which all the imaginary elements are zeros, with the real-to-complex `scfft1d/zdfft1d` FFT applied to the same vector. The advantage of the latter approach is that only half of the input data storage is required and there is no need to zero the imaginary part. The last two columns are stored in the real array of size `(n+2)` containing the complex conjugate-symmetric vector in CCS format.

To compute a forward FFT of a given length, first initialize the coefficients by calling the routine you are going to use with *`isign`* `= 0`. Thereafter, any number of real-to-complex and complex-to-real transforms of the same length can be computed by calling that routine with the *`isign`* value other than `0`.

Table 12-2 Comparison of the Storage Effects of Complex-to-Complex and Real-to-Complex FFTs

Input Vectors			Output Vectors			
`cfft1d`		`scfft1d`	`cfft1d`		`scfft1d`	
Complex Data		Real Data	Complex Data		Real Data	
Real	Imaginary		Real	Imaginary	(Real)	(Imaginary)
0.841471	0.000000	0.841471	1.543091	0.000000	1.543091	0.000000
0.909297	0.000000	0.909297	3.875664	0.910042	3.875664	0.910042
0.141120	0.000000	0.141120	-0.915560	-0.397326	-0.915560	-0.397326
-0.756802	0.000000	-0.756802	-0.274874	-0.121691	-0.274874	-0.121691
-0.958924	0.000000	-0.958924	-0.181784	0.000000	-0.181784	0.000000
-0.279415	0.000000	-0.279415	-0.274874	0.121691		
0.656987	0.000000	0.656987	-0.915560	0.397326		
0.989358	0.000000	0.989358	3.875664	-0.910042		

scfft1d/dzfft1d

Fortran-interface routines. Compute forward FFT of a real vector and represent the complex conjugate-symmetric result in CCS format (in-place).

Syntax

```
call scfft1d ( r, n, isign, wsave )
```

```
call dzfft1d ( r, n, isign, wsave )
```

Description

The operation performed by the `scfft1d/dzfft1d` routines is determined by the value of *isign*. See the equations of the operations for Real-to-Complex One-dimensional FFTs above. These routines are complementary to the complex-to-real transform routines `csfft1d/zdfft1d`.

Input Parameters

r — `REAL` for `scfft1d`
`DOUBLE PRECISION` for `dzfft1d`

Array, `DIMENSION` at least (*n*+2). First *n* elements contain the input vector to be transformed. The elements *r*(*n*+1) and *r*(*n*+2) are used on output. The array *r* is not referenced if *isign* = 0.

n — `INTEGER`. Transform length; *n* must be a power of 2.

isign — `INTEGER`. Flag indicating the type of operation to be performed:
if *isign* is 0, initialize the coefficients *wsave*;
if *isign* is not 0, perform the forward FFT.

wsave — `REAL` for `scfft1d`
`DOUBLE PRECISION` for `dzfft1d`

Array, `DIMENSION` at least (2***n*+4). If *isign* = 0, then *wsave* contains output data. Otherwise, *wsave* contains coefficients required to perform the FFT that has been initialized on a previous call to this routine or the complementary complex-to-real FFT routine.

Output Parameters

r — If *isign* = 0, *r* does not change. If *isign* is not 0, the output real-valued array *r*(1:*n*+2) contains the complex conjugate-symmetric vector *z*(1:*n*) packed in CCS format for Fortran interface.
The table below shows the relationship between them.

r(1)	r(2)	r(3)	r(4)	...	r(n-1)	r(n)	r(n+1)	r(n+2)
z(1)	0	REz(2)	IMz(2)	...	REz(n/2)	IMz(n/2)	z(n/2+1)	0

The full complex vector *z*(1:*n*) is defined by

```
z(i) = cmplx(r(2*i-1), r(2*i)),
1 ≤ i ≤ n/2+1,
```

`z(n/2+i) = conjg(z(n/2+2-i)),`
`2 ≤ i ≤ n/2.`

Then, `z(1:n)` is the forward FFT of a real input vector `r(1:n)`.

wsave If *isign* = 0, *wsave* contains the coefficients required by the called routine. Otherwise *wsave* does not change.

scfft1dc/dzfft1dc

C-interface routines. Compute forward FFT of a real vector and represent the complex conjugate-symmetric result in CCS format (in-place).

Syntax

```
void scfft1dc ( float* r, int n, int isign, float* wsave );
void dzfft1dc ( double* r, int n, int isign, double* wsave );
```

Description

The operation performed by the `scfft1dc/dzfft1dc` routines is determined by the value of *isign*. See the equations of the operations for the Real-to-Complex One-dimensional FFTs above.
These routines are complementary to the complex-to-real transform routines `csfft1dc/zdfft1dc`.

Input Parameters

r `float*` for `scfft1dc`
`double*` for `dzfft1dc`

Pointer to an array of size at least (*n*+2). First *n* elements contain the input vector to be transformed. The array *r* is not referenced if *isign* = 0.

n `int`. Transform length; *n* must be a power of 2.

isign `int`. Flag indicating the type of operation to be performed:

if *isign* is 0, initialize the coefficients *wsave*;

if *isign* is not 0, perform the forward FFT.

wsave `float*` for `scfft1dc`
`double*` for `dzfft1dc`

Pointer to an array of size at least `(2*n+4)`.
If `isign = 0`, then *wsave* contains output data. Otherwise, *wsave* contains coefficients required to perform the FFT that has been initialized on a previous call to this routine or the complementary complex-to-real FFT routine.

Output Parameters

r If *isign* = 0, *r* does not change. If *isign* is not 0, the output real-valued array `r(0:n+1)` contains the complex conjugate-symmetric vector `z(0:n-1)` packed in CCS format for C-interface.
The table below shows the relationship between them.

r(0)	r(1)	r(2)	...	r(n/2)	r(n/2+1)	r(n/2+2)	...	r(n)	r(n+1)
z(0)	REz(1)	REz(2)	...	z(n/2)	0	IMz(1)	...	IMz(n/2-1)	0

The full complex vector `z(0:n-1)` is defined by

$z(i) = \mathrm{cmplx}(r(i), r(n/2+1+i)),\ 0 \le i \le n/2,$

$z(n/2+i) = \mathrm{conjg}(z(n/2-i)),\ 1 \le i \le n/2-1.$
Then, `z(0:n-1)` is the forward FFT of the real input vector of length *n*.

wsave If *isign* = 0, *wsave* contains the coefficients required by the called routine. Otherwise *wsave* does not change.

Complex-to-Real One-dimensional FFTs

Each of the complex-to-real routines computes a one-dimensional inverse FFT according to the mathematical equation

$$t_j = \frac{1}{n}\sum_{k=0}^{n-1} z_k * w^{j*k},\quad 0 \le j \le n-1$$

The mathematical input is the complex conjugate-symmetric vector z_j, $0 \le j \le n-1$, , where `z(n/2+i) = conjg(z(n/2-i))`, $1 \le i \le n/2-1$, and moreover `z(0)` and `z(n/2)` are real values.

The mathematical result is $t_j = \mathrm{cmplx}(r_j, 0)$, where r_j is a real vector, $0 \le j \le n-1$.

Input to the complex-to-real transform routines is a real array of size `(n+2)`, which contains the complex conjugate-symmetric vector `z(0:n-1)` in CCS format (see Real-to-Complex One-dimensional FFTs above).

Output of the complex-to-real routines is a real vector of size *n*.

Table 12-3 is identical to Table 12-2, except for reversing the input and output vectors. In the complex-to-real routines the last two columns are stored in the input real array of size `(n+2)` containing the complex conjugate-symmetric vector in CCS format.

To compute an inverse FFT of a given length, first initialize the coefficients by calling the routine you are going to use with *isign* = 0. Thereafter, any number of real-to-complex and complex-to-real transforms of the same length can be computed by calling the appropriate routine with the *isign* value other than 0.

Table 12-3 Comparison of the Storage Effects of Complex-to-Real and Complex-to-Complex FFTs

Output Vectors			Input Vectors			
cfft1d		**csfft1d**	**cfft1d**		**csfft1d**	
Complex Data		**Real Data**	**Complex Data**		**Real Data**	
Real	**Imaginary**		**Real**	**Imaginary**	**(Real)**	**(Imaginary)**
0.841471	0.000000	0.841471	1.543091	0.000000	1.543091	0.000000
0.909297	0.000000	0.909297	3.875664	0.910042	3.875664	0.910042
0.141120	0.000000	0.141120	-0.915560	-0.397326	-0.915560	-0.397326
-0.756802	0.000000	-0.756802	-0.274874	-0.121691	-0.274874	-0.121691
-0.958924	0.000000	-0.958924	-0.181784	0.000000	-0.181784	0.000000
-0.279415	0.000000	-0.279415	-0.274874	0.121691		
0.656987	0.000000	0.656987	-0.915560	0.397326		
0.989358	0.000000	0.989358	3.875664	-0.910042		

csfft1d/zdfft1d

Fortran-interface routines. Compute inverse FFT of a complex conjugate-symmetric vector packed in CCS format (in-place).

Syntax

```
call csfft1d ( r, n, isign, wsave )
call zdfft1d ( r, n, isign, wsave )
```

Description

The operation performed by the `csfft1d/zdfft1d` routines is determined by the value of *isign*. See the equations of the operations for the Complex-to-Real One-dimensional FFTs above.

These routines are complementary to the real-to-complex transform routines `scfft1d/dzfft1d`.

Input Parameters

r

`REAL` for `csfft1d`
`DOUBLE PRECISION` for `zdfft1d`

Array, `DIMENSION` at least (*n*+2).
Not referenced if *isign* = 0.

If *isign* is not 0, then *r*(1:*n*+2) contains the complex conjugate-symmetric vector packed in CCS format for Fortran-interface.
The table below shows the relationship between them.

r(1)	r(2)	r(3)	r(4)	...	r(n-1)	r(n)	r(n+1)	r(n+2)
z(1)	0	REz(2)	IMz(2)	...	REz(n/2)	IMz(n/2)	z(n/2+1)	0

The full complex vector *z*(1:*n*) is defined by

```
z(i) = cmplx(r(2*i-1), r(2*i)),
1 ≤ i ≤ n/2+1,

z(n/2+i) = conjg(z(n/2+2-i)),
2 ≤ i ≤ n/2.
```

After the transform, *r*(1:*n*) contains the inverse FFT of the complex conjugate-symmetric vector *z*(1:*n*).

n INTEGER. Transform length; *n* must be a power of 2.

isign INTEGER. Flag indicating the type of operation to be performed:
if *isign* is 0, initialize the coefficients wsave;
if *isign* is not 0, perform the inverse FFT.

wsave REAL for csfft1d
DOUBLE PRECISION for zdfft1d
Array, DIMENSION at least (2**n*+4). If *isign* = 0, then *wsave* contains output data. Otherwise, *wsave* contains coefficients required to perform the FFT that has been initialized on a previous call to this routine or the complementary real-to-complex FFT routine.

Output Parameters

r If *isign* is not 0, then *r*(1:*n*) is the real result of the inverse FFT of the complex conjugate-symmetric vector *z*(1:*n*). Does not change if *isign* = 0.

wsave If *isign* = 0, *wsave* contains the coefficients required by the called routine. Otherwise *wsave* does not change.

csfft1dc/zdfft1dc

C-interface routines.Compute inverse FFT of a complex conjugate-symmetric vector packed in CCS format (in-place).

Syntax

```
void csfft1dc ( float* r, int n, int isign, float* wsave )
void zdfft1dc ( double* r, int n, int isign, double* wsave )
```

Description

The operation performed by the csfft1dc/zdfft1dc routines is determined by the value of *isign*. See the equations of the operations for the Complex-to-Real One-dimensional FFTs above.

These routines are complementary to the real-to-complex transform routines `scfft1dc/dzfft1dc`.

Input Parameters

r

`float*` for `csfft1dc`
`double*` for `zdfft1dc`

Pointer to an array of size at least `(`*n*`+2)`. Not referenced if *isign* = `0`.

If *isign* is not `0`, then *r*`(0:`*n*`+1)` contains the complex conjugate-symmetric vector packed in CCS format for C-interface.
The table below shows the relationship between them.

r(0)	*r*(1)	*r*(2)	...	*r*(*n*/2)	*r*(*n*/2+1)	*r*(*n*/2+2)	...	*r*(*n*)	*r*(*n*+1)
z(0)	RE*z*(1)	RE*z*(2)	...	*z*(*n*/2)	0	IM*z*(1)	...	IM*z*(*n*/2-1)	0

The full complex vector *z*`(0:`*n*`-1)` is defined by
z`(i)` = `cmplx(`*r*`(i),`*r*`(n/2+1+i))`, `0` ≤ `i` ≤ *n*`/2`,

z`(`*n*`/2+i)` = `conjg(`*z*`(`*n*`/2-i))`, `1` ≤ `i` ≤ *n*`/2-1`.
After the transform, *r*`(0:`*n*`-1)` is the inverse FFT of the complex conjugate-symmetric vector *z*`(0:`*n*`-1)`.

n

`int`. Transform length; *n* must be a power of 2.

isign

`int`. Flag indicating the type of operation to be performed:
if *isign* = `0`, initialize the coefficients *wsave*;
if *isign* is not `0`, perform the inverse FFT.

wsave

`float*` for `csfft1dc`
`double*` for `zdfft1dc`
Pointer to an array of size at least `(2*`*n*`+4)`.
If *isign* `= 0`, then *wsave* contains output data. Otherwise, *wsave* contains coefficients required to perform the FFT that has been initialized on a previous call to this routine or the complementary real-to-complex FFT routine.

Output Parameters

r

If *isign* is not `0`, then *r*`(0:`*n*`-1)` is the real result of the inverse FFT of the complex conjugate-symmetric vector *z*`(0:`*n*`-1)`. Does not change if *isign* = `0`.

wsave

If *isign* = `0`, *wsave* contains the coefficients required by the called routine. Otherwise *wsave* does not change.

Two-dimensional FFTs

The two-dimensional FFTs are functionally the same as one-dimensional FFTs. They contain the following groups:

- Complex-to-Complex Transforms
- Real-to-Complex Transforms
- Complex-to-Real Transforms.

All two-dimensional FFTs are in-place. Transform lengths must be a power of 2. The complex-to-complex transform routines perform both forward and inverse transforms of a complex matrix. The real-to-complex transform routines perform forward transforms of a real matrix. The complex-to-real transform routines perform inverse transforms of a complex conjugate-symmetric matrix, which is packed in a real array.

The naming conventions are also the same as those for one-dimensional FFTs, with "2d" replacing "1d" in all cases. Table 12-4 lists the two-dimensional FFT routine groups and the data types associated with them.

Table 12-4 Two-dimensional FFTs: Names and Data Types

Group	Stored as FORTRAN Complex Data	Stored as C Real Data	Data Types	Description
Complex-to-Complex	`cfft2d/` `zfft2d`	`cfft2dc/` `zfft2dc`	c, z	Transform complex data to complex data.
Real-to-Complex	`scfft2d/` `dzfft2d`	`scfft2dc/` `dzfft2dc`	sc, dz	Transform forward real-to-complex data. Complement `csfft2d/zdfft2d` and `csfft2dc/zdfft2dc` FFTs.
Complex-to-Real	`csfft2d/` `zdfft2d`	`csfft2dc/` `zdfft2dc`	cs, zd	Transform inverse complex-to-real data. Complement `scfft2d/dzfft2d` and `scfft2dc/dzfft2dc` FFTs.

The C-interface requires scalar values to be passed by value. The major difference between the one-dimensional and two-dimensional FFTs is that your application does not need to provide storage for transform coefficients.

The data storage types and data structure requirements are the same as for one-dimensional FFTs. For more information, see the Data Storage Types and Data Structure Requirements sections at the beginning of this chapter.

Complex-to-Complex Two-dimensional FFTs

Each of the complex-to-complex routines computes a forward or inverse FFT of a complex matrix in-place.

The forward FFT is computed according to the mathematical equation

$$z_{i,j} = \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} r_{k,l} * w_m^{-i*k} * w_n^{-j*l}, \quad 0 \le i \le m-1, \quad 0 \le j \le n-1$$

The inverse FFT is computed according to the mathematical equation

$$r_{i,j} = \frac{1}{m*n} \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} z_{k,l} * w_m^{i*k} * w_n^{j*l}, \quad 0 \le i \le m-1, \quad 0 \le j \le n-1$$

where $w_m = \exp\left[\frac{2\pi i}{m}\right]$, $w_n = \exp\left[\frac{2\pi i}{n}\right]$, *i* being the imaginary unit.

The operation performed by the complex-to-complex routines is determined by the value of the *isign* parameter.

If *isign* = -1, perform the forward FFT where input and output are in normal order.
If *isign* = +1, perform the inverse FFT where input and output are in normal order.
If *isign* = -2, perform the forward FFT where input is in normal order and output is in bit-reversed order.
If *isign* = +2, perform the inverse FFT where input is in bit-reversed order and output is in normal order.

The above equations apply to all FFTs with all data types indicated in Table 12-4.

cfft2d/zfft2d

Fortran-interface routines. Compute the forward or inverse FFT of a complex matrix (in-place).

Syntax

```
call cfft2d ( r, m, n, isign )
call zfft2d ( r, m, n, isign )
```

Description

The operation performed by the `cfft2d/zfft2d` routines is determined by the value of *`isign`*. See the equations of the operations for Complex-to-Complex Two-dimensional FFTs.

Input Parameters

r	`COMPLEX` for `cfft2d` `DOUBLE COMPLEX` for `zfft2d` Array, `DIMENSION` at least (*m*,*n*), with its leading dimension equal to *m*. This array contains the complex matrix to be transformed.
m	`INTEGER`. Column transform length (number of rows); *m* must be a power of 2.
n	`INTEGER`. Row transform length (number of columns); *n* must be a power of 2.
isign	`INTEGER`. Flag indicating the type of operation to be performed: if *isign* = `-1`, perform the forward FFT where input and output are in normal order; if *isign* = `+1`, perform the inverse FFT where input and output are in normal order; if *isign* = `-2`, perform the forward FFT where input is in normal order and output is in bit-reversed order; if *isign* = `+2`, perform the inverse FFT where input is in bit-reversed order and output is in normal order.

Output Parameters

r	Contains the complex result of the transform depending on *isign*.

cfft2dc/zfft2dc

C-interface routines. Compute the forward or inverse FFT of a complex matrix (in-place).

Syntax

```
void cfft2dc ( float* r, float* i, int m, int n, int isign )
void zfft2dc ( double* r, double* i, int m, int n, int isign )
```

Description

The operation performed by the `cfft2dc/zfft2dc` routines is determined by the value of *`isign`*. See the equations of the operations for the Complex-to-Complex Two-dimensional FFTs above.

Input Parameters

r — `float*` for `cfft2dc`
`double*` for `zfft2dc`

Pointer to a two-dimensional array of size at least (*m*,*n*), with its leading dimension equal to *n*. The array contains the real parts of a complex matrix to be transformed.

i — `float*` for `cfft2dc`
`double*` for `zfft2dc`

Pointer to a two-dimensional array of size at least (*m*,*n*), with its leading dimension equal to *n*. The array contains the imaginary parts of a complex matrix to be transformed.

m — `int`. Column transform length (number of rows); *m* must be a power of 2.

n — `int`. Row transform length (number of columns); *n* must be a power of 2.

isign — `int`. Flag indicating the type of operation to be performed:

if *isign* = `-1`, perform the forward FFT where input and output are in normal order;
if *isign* = `+1`, perform the inverse FFT where input and output are in normal order;
if *isign* = `-2`, perform the forward FFT where input is in normal order and

output is in bit-reversed order;
if *isign* = +2, perform the inverse FFT where input is in bit-reversed order and output is in normal order.

Output Parameters

r Contains the real parts of the complex result depending on *isign*.

i Contains the imaginary parts of the complex depending on *isign*.

Real-to-Complex Two-dimensional FFTs

Each of the real-to-complex routines computes the forward FFT of a real matrix according to the mathematical equation

$$z_{i,j} = \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} t_{k,l} * w_m^{-i*k} * w_n^{-j*l}, \quad 0 \le i \le m-1, \ 0 \le j \le n-1$$

$t_{k,l}$ = cmplx($r_{k,l}$,0), where $r_{k,l}$ is a real input matrix, $0 \le k \le m-1$, $0 \le l \le n-1$.
The mathematical result $z_{i,j}$, $0 \le i \le m-1$, $0 \le j \le n-1$, is the complex matrix of size (m,n). Each column is the complex conjugate-symmetric vector as follows:

for $0 \le j \le n-1$,

$z(m/2+i,j) = conjg(z(m/2-i,j))$, $1 \le i \le m/2-1$.
Moreover, $z(0,j)$ and $z(m/2,j)$ are real values for j=0 and j=*n*/2.

This mathematical result can be stored in the real two-dimensional array of size (*m*+2,*n*+2) or in the complex two-dimensional array of size (*m*/2+1,*n*+1) for Fortran-interface and in the complex two-dimensional array of size (*m*+1,*n*/2+1) for C-interface. The data storage of CCS format is defined later for Fortran-interface and C-interface routines separately.

scfft2d/dzfft2d

Fortran-interface routines. Compute forward FFT of a real matrix and represent the complex conjugate-symmetric result in CCS format (in-place).

Syntax

```
call scfft2d ( r, m, n )
call dzfft2d ( r, m, n )
```

Description

See the equations of the operations for the Real-to-Complex Two-dimensional FFTs above.

These routines are complementary to the complex-to-real transform routines `csfft2d/zdfft2d`.

Input Parameters

r	`REAL` for `scfft2d` `DOUBLE PRECISION` for `dzfft2d` Array, `DIMENSION` at least `(`*m*`+2,`*n*`+2)`, with its leading dimension equal to `(`*m*`+2)`. The first *m* rows and *n* columns of this array contain the real matrix to be transformed. Table 12-5 presents the input data layout.
m	`INTEGER`. Column transform length (number of rows); *m* must be a power of 2.
n	`INTEGER`. Row transform length (number of columns); *n* must be a power of 2.

Table 12-5 Fortran-interface Real Data Storage for the Real-to-Complex and Complex-to-Real Two-dimensional FFTs

r(1,1)	r(1,2)	...	r(1,n-1)	r(1,n)	n/u	n/u
r(2,1)	r(2,2)	...	r(2,n-1)	r(2,n)	n/u	n/u
r(3,1)	r(3,2)	...	r(3,n-1)	r(3,n)	n/u	n/u
r(4,1)	r(4,2)	...	r(4,n-1)	r(4,n)	n/u	n/u
...	...	...	...	...	...	...
r(m-1,1)	r(m-1,2)	...	r(m-1,n-1)	r(m-1,n)	n/u	n/u
r(m,1)	r(m,2)	...	r(m,n-1)	r(m,n)	n/u	n/u
n/u	n/u	...	n/u	n/u	n/u	n/u
n/u	n/u	...	n/u	n/u	n/u	n/u

* n/u - not used

Output Parameters

r The output real array `r(1:m+2,1:n+2)` contains the complex conjugate-symmetric matrix `z(1:m,1:n)` packed in CCS format for Fortran-interface as follows:

- Rows `1` and `m+1` contain in `n+2` locations the complex conjugate-symmetric vectors `z(1,j)` and `z(m/2+1,j)` packed in CCS format (seeReal-to-Complex One-dimensional FFTs above).

  The full complex vector `z(1,j)` is defined by:

  `z(1,j) = cmplx(r(1,2*j-1),r(1,2*j))`, $1 \le j \le n/2+1$,

  `z(1,n/2+1+j) = conjg(z(1,n/2+1-j))`, $1 \le j \le n/2-1$.

  The full complex vector `z(m/2+1,j)` is defined by:

  `z(m/2+1,j) = cmplx(r(m+1,2*j-1),r(m+1,2*j))`,
  $1 \le j \le n/2+1$,

  `z(m/2+1,n/2+1+j) = conjg(z(m/2+1,n/2+1-j))`,
  $1 \le j \le n/2-1$;

- Rows from `3` to `m` contain in `n` locations complex vectors represented as

  `z(i+1,j) = cmplx(r(2*i+1,j),r(2*i+2,j))`,
  $1 \le i \le m/2-1$, $1 \le j \le n$.

- The rest matrix elements can be obtained from
  `z(m/2+1+i,j) = conjg(z(m/2+1-i,j))`,
  `1 ≤ i ≤ m/2-1, 1 ≤ j ≤ n`.

The storage of the complex conjugate-symmetric matrix *z* for Fortran-interface is shown in Table 12-6.

Table 12-6 Fortran-interface Data Storage of CCS Format for the Real-to-Complex and Complex-to-Real Two-Dimensional FFTs

z(1,1)	0	REz(1,2)	IMz(1,2)	...	REz(1,n/2)	IMz(1,n/2)	z(1, n/2+1)	0
0	0	0	0	...	0	0	0	0
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)	n/u	n/u
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)	n/u	n/u
...	...	...	...	...	...	...	n/u	n/u
REz(m/2,1)	REz(m/2,2)	REz(m/2,3)	REz(m/2,4)	...	REz(m/2, n-1)	REz(m/2, n)	n/u	n/u
IMz(m/2,1)	IMz(m/2,2)	IMz(m/2,3)	IMz(m/2,4)	...	IMz(m/2, n-1)	IMz(m/2, n)	n/u	n/u
z(m/2+1,1)	0	REz(m/2+1,2)	IMz(m/2+1,2)	...	REz(m/2+1, n/2)	IMz(m/2+1, n/2)	z(m/2+1, n/2+1)	0
0	0	0	0	...	0	0	n/u	n/u

* n/u - not used

scfft2dc/dzfft2dc

C-interface routine. Compute forward FFT of a real matrix and represent the complex conjugate-symmetric result in CCS format (in-place).

Syntax

```
void scfft2dc ( float* r, int m, int n )
void dzfft2dc ( double* r, int m, int n )
```

Description

See the equations of the operations for the Real-to-Complex Two-dimensional FFTs above.

These routines are complementary to the complex-to-real transform routines `csfft2dc/zdfft2dc`.

Input Parameters

r — `float*` for `scfft2dc`
`double*` for `dzfft2dc`

Pointer to an array of size at least `(`*m*`+2,`*n*`+2)`, with its leading dimension equal to `(`*n*`+2)`. The first *m* rows and *n* columns of this array contain the real matrix to be transformed.

Table 12-7 presents the input data layout.

m — `int`. Column transform length;
m must be a power of 2.

n — `int`. Row transform length;
n must be a power of 2.

Table 12-7 C-interface Real Data Storage for a Real-to-Complex and Complex-to-Real Two-dimensional FFTs

r(0,0)	r(0,1)	...	r(0,n-2)	r(0,n-1)	n/u	n/u
r(1,0)	r(1,1)	...	r(1,n-2)	r(1,n-1)	n/u	n/u
r(2,0)	r(2,1)	...	r(2,n-2)	r(2,n-1)	n/u	n/u
r(3,0)	r(3,1)	...	r(3,n-2)	r(3,n-1)	n/u	n/u
...	...	...	...	...	...	...
r(m-2,0)	r(m-2,1)	...	r(m-2,n-2)	r(m-2,n-1)	n/u	n/u
r(m-1,0)	r(m-1,1)	...	r(m-1,n-2)	r(m-1,n-1)	n/u	n/u
n/u	n/u	...	n/u	n/u	n/u	n/u
n/u	n/u	...	n/u	n/u	n/u	n/u

Output Parameters

r — The output real array `r(0:`*m*`+1,0:`*n*`+1)` contains the complex conjugate-symmetric matrix `z(0:`*m*`-1,0:`*n*`-1)` packed in CCS format for C-interface as follows:

- Columns `0` and $n/2$ contain in $m+2$ locations the complex conjugate-symmetric vectors $z(i,0)$ and $z(i,n/2)$ in CCS format (seeReal-to-Complex One-dimensional FFTs above).

  The full complex vector $z(i,0)$ is defined by:

  $z(i,0) = \mathrm{cmplx}(r(i,0), r(m/2+i+1,0)),\ 0 \le i \le m/2,$

  $z(m/2+i,0) = \mathrm{conjg}(z(m/2-i,0)),\ 1 \le i \le m/2-1.$

  The full complex vector $z(i,n/2)$ is defined by:

  $z(i,n/2) = \mathrm{cmplx}(r(i,n/2), r(m/2+i+1,n/2)),\ 0 \le i \le m/2,$

  $z(m/2+i,n/2) = \mathrm{conjg}(z(m/2-i,n/2)),\ 1 \le i \le m/2-1.$

- Columns from `1` to $n/2-1$ contain real parts, and columns from $n/2+2$ to n contain imaginary parts of complex vectors. These values for each vector are stored in m locations represented as follows

  $z(i,j) = \mathrm{cmplx}(r(i,j), r(i,n/2+1+j)),$
  $0 \le i \le m-1,\ 1 \le j \le n/2-1.$

- The rest matrix elements can be obtained from

  $z(i,n/2+j) = \mathrm{conjg}(z(i,n/2-j)),$
  $0 \le i \le m-1,\ 1 \le j \le n/2-1.$

The storage of the complex conjugate-symmetric matrix z for C-interface is shown in Table 12-8.

Table 12-8 C-interface Data Storage of CCS Format for the Real-to-Complex and Complex-to-Real Two-dimensional FFT

z(0,0)	REz(0,1)	...	REz(0, n/2-1)	z(0,n/2)	0	IMz(0,1)	...	IMz(0, n/2-1)	0
REz(1,0)	REz(1,1)	...	REz(1, n/2-1)	REz(1,n/2)	0	IMz(1,1)	...	IMz(1, n/2-1)	0
...	...	...	...	...	0	...	...	...	0
REz(m/2-1, 0)	REz(m/2-1, 1)	...	REz(m/2-1, n/2-1)	REz(m/2-1, n/2)	0	IMz(m/2-1, 1)	...	IMz(m/2-1, n/2-1)	0
z(m/2,0)	REz(m/2,1)	...	REz(m/2, n/2-1)	z(m/2,n/2)	0	IMz(m/2,1)	...	IMz(m/2, n/2-1)	0
0	REz(m/2+1, 1)	...	REz(m/2+1, n/2-1)	0	0	IMz(m/2+1, 1)	...	IMz(m/2+1, n/2-1)	0
IMz(1,0)	REz(m/2+2, 1)	...	REz(m/2+2, n/2-1)	IMz(1,n/2)	0	IMz(m/2+2, 1)	...	IMz(m/2+2, n/2-1)	0
...	...	...	...	...	0	...	...	...	0
IMz(m/2-2, 0)	REz(m-1,1)	...	REz(m-1, n/2-1)	IMz(m/2-2, n/2)	0	IMz(m-1,1)	...	IMz(m-1, n/2-1)	0
IMz(m/2-1, 0)	n/u	...	n/u	IMz(m/2-1, n/2)	n/u	n/u	...	n/u	n/u
0	n/u	...	n/u	0	n/u	n/u	...	n/u	n/u

Complex-to-Real Two-dimensional FFTs

Each of the complex-to-real routines computes a two-dimensional inverse FFT according to the mathematical equation:

$$t_{i,j} = \frac{1}{m*n}\sum_{k=0}^{m-1}\sum_{l=0}^{n-1} z_{k,l}*w_m^{i*k}*w_n^{j*l}, \quad 0 \le i \le m-1, \quad 0 \le j \le n-1$$

The mathematical input $z_{i,j}$, $0 \le i \le m-1$, $0 \le j \le n-1$, is a complex matrix of size (m,n). Each column is the complex conjugate-symmetric vector as follows:

for $0 \le j \le n-1$,
$z(m/2+i,j) = conjg(z(m/2-i,j))$, $1 \le i \le m/2-1$.
Moreover, $z(0,j)$ and $z(m/2,j)$ are real values for $j=0$ and $j=n/2$.

This mathematical result can be stored in the real two-dimensional array of size `(m+2,n+2)` or in the complex two-dimensional array of size `(m/2+1,n+1)` for Fortran-interface and in the complex two-dimensional array of size `(m+1,n/2+1)` for C-interface. The data storage of CCS format is defined later for Fortran-interface and C-interface routines separately.

The mathematical result of the transform is $t_{k,l} = cmplx(r_{k,l},0)$, where $r_{k,l}$ is the real matrix, $0 \le k \le m-1$, $0 \le l \le n-1$.

csfft2d/zdfft2d

Fortran-interface routine. Compute inverse FFT of a complex conjugate-symmetric matrix packed in CCS format (in-place).

Syntax

```
call csfft2d ( r, m, n )
call zdfft2d ( r, m, n )
```

Description

See the equations of the operations for the Complex-to-Real Two-dimensional FFTs above. These routines are complementary to the real-to-complex transform routines `scfft2d/dzfft2d`.

Input Parameters

r — `SINGLE PRECISION REAL*4` for `csfft2d`
`DOUBLE PRECISION REAL*8` for `zdfft2d`

Array, `DIMENSION` at least `(m+2,n+2)`, with its leading dimension equal to `(m+2)`. This array contains the complex conjugate-symmetric matrix in CCS format to be transformed. The input data layout is given in Table 12-6.

m — `INTEGER`. Column transform length (number of rows); *m* must be a power of 2.

n — `INTEGER`. Row transform length (number of columns); *n* must be a power of 2.

Output Parameters

r — Contains the real result returned by the transform. For the output data layout, see Table 12-5.

csfft2dc/zdfft2dc

C-interface routines. Compute inverse FFT of a complex conjugate-symmetric matrix packed in CCS format (in-place).

Syntax

```
void csfft2dc ( float* r, int m, int n );
void zdfft2dc ( double* r, int m, int n );
```

Description

See the equations of the operations for the Complex-to-Real Two-dimensional FFTs above. These routines are complementary to the real-to-complex transform routines `scfft2dc/dzfft2dc`.

Input Parameters

r — `float*` for `csfft2dc`
`double*` for `zdfft2dc`

Pointer to an array of size at least `(`*m*`+2,`*n*`+2)`, with its leading dimension equal to `(`*n*`+2)`. This array contains the complex conjugate-symmetric matrix in CCS format to be transformed. The input data layout is given in Table 12-8.

m — `int`. Column transform length; *m* must be a power of 2.

n — `int`. Row transform length; *n* must be a power of 2.

Output Parameters

r — Contains the real result returned by the transform. The output data layout is the same as that for the input data of `scfft2dc/dzfft2dc`. See Table 12-7 for the details.

Linear Solvers Basics

A

Many applications in science and engineering require the solution of a system of linear equations. This problem is usually expressed mathematically by the matrix-vector equation, $Ax = b$, where *A* is an *n* by *n* matrix and x and b are *n* element column vectors. The matrix *A* is usually referred to as the coefficient matrix, and the vectors x and b are referred to as the solution vector and the right-hand side, respectively.

In many real-life applications, most of the elements in *A* are zero. Such a matrix is referred to as sparse. Conversely, matrices with very few zero elements are called dense. For sparse matrices, computing the solution to the equation $Ax = b$ can be made much more efficient with respect to both storage and computation time, if the sparsity of the matrix can be exploited. The more an algorithm can exploit the sparsity without sacrificing the correctness, the better the algorithm.

Generally speaking, computer software that finds solutions to systems of linear equations is called a solver. A solver designed to work specifically on sparse systems of equations is called a sparse solver. Solvers are usually classified into two groups - direct and iterative.

Iterative Solvers start with an initial approximation to a solution and attempt to estimate the difference between the approximation and the true result. Based on the difference, an iterative solver calculates a new approximation that is closer to the true result than the initial approximation. This process is repeated until the difference between the approximation and the true result is sufficiently small. The main drawback to iterative solvers is that the rate of convergence depends greatly on the values in the matrix *A*. Consequently, it is not possible to predict how long it will take for an iterative solver to produce a solution. In fact, for ill-conditioned matrices, the iterative process will not converge to a solution at all. However, for well-conditioned matrices it is possible for iterative solvers to converge to a solution very quickly. Consequently for the right applications, iterative solvers can be very efficient.

Direct Solvers, on the other hand, often factor the matrix *A* into the product of two triangular matrices and then perform a forward and backward triangular solve.

This approach makes the time required to solve a systems of linear equations relatively predictable, based on the size of the matrix. In fact, for sparse matrices, the solution time can be predicted based on the number of non-zero elements in the array A.

Matrix Fundamentals

A matrix is a rectangular array of either real or complex numbers. A matrix is denoted by a capital letter; its elements are denoted by the same lower case letter with row/column subscripts. Thus, the value of the element in row i and column j in matrix A is denoted by $a(i,j)$.
For example, a 3 by 4 matrix A, is written as follows:

$$A = \begin{bmatrix} a(1,1) & a(1,2) & a(1,3) & a(1,4) \\ a(2,1) & a(2,2) & a(2,3) & a(2,4) \\ a(3,1) & a(3,2) & a(3,3) & a(3,4) \end{bmatrix}$$

Note that with the above notation, we assume the standard Fortran programming language convention of starting array indices at 1 rather than the C programming language convention of starting them at 0.

A matrix in which all of the elements are real numbers is called a real matrix. A matrix that contains at least one complex number is called a complex matrix.
A real or complex matrix A with the property that $a(i,j) = a(j,i)$, is called a symmetric matrix. A complex matrix A with the property that $a(i,j) = conj(a(j,i))$, is called a Hermitian matrix. Note that programs that manipulate symmetric and Hermitian matrices need only store half of the matrix values, since the values of the non-stored elements can be quickly reconstructed from the stored values.

A matrix that has the same number of rows as it has columns is referred to as a square matrix. The elements in a square matrix that have same row index and column index are called the diagonal elements of the matrix, or simply the diagonal of the matrix.

The transpose of a matrix A is the matrix obtained by “flipping” the elements of the array about its diagonal. That is, we exchange the elements $a(i,j)$ and $a(j,i)$. For a complex matrix, if we both flip the elements about the diagonal and then take the complex conjugate of the element, the resulting matrix is called the Hermitian transpose or conjugate transpose of the original matrix. The transpose and Hermitian transpose of a matrix A are denoted by A^T and A^H respectively.

A column vector, or simply a vector, is a $n \times 1$ matrix, and a row vector is a $1 \times n$ matrix.
A real or complex matrix A is said to be positive definite if the vector-matrix product $x^T A x$ is greater than zero for all non-zero vectors x. A matrix that is not positive definite is referred to as indefinite.

An upper (or lower) triangular matrix, is a square matrix in which all elements below (or above) the diagonal are zero. A unit triangular matrix is an upper or lower triangular matrix with all 1's along the diagonal.

A matrix P is called a permutation matrix if, for any matrix A, the result of the matrix product PA is identical to A except for interchanging the rows of A. For a square matrix, it can be shown that if PA is a permutation of the rows of A, then AP^T is the same permutation of the columns of A. Additionally, it can be shown that the inverse of P is P^T.

In order to save space, a permutation matrix is usually stored as a linear array, called a permutation vector, rather than as an array. Specifically, if the permutation matrix maps the `i`-th row of a matrix to the `j`-th row, then the `i`-th element of the permutation vector is `j`.

A matrix with non-zero elements only on the diagonal is called a diagonal matrix. As is the case with a permutation matrix, it is usually stored as a vector of values, rather than as a matrix.

Direct Method

For solvers that use the direct method, the basic technique employed in finding the solution of the system `Ax = b` is to first factor A into triangular matrices. That is, find a lower triangular matrix L and an upper triangular matrix U, such that $A = LU$. Having obtained such a factorization (usually referred to as an LU decomposition or LU factorization), the solution to the original problem can be rewritten as follows.

$$
\begin{aligned}
& Ax = b \\
\Rightarrow \quad & LUx = b \\
\Rightarrow \quad & (Ux) = b
\end{aligned}
$$

This leads to the following two-step process for finding the solution to the original system of equations:

1. Solve the systems of equations `Ly = b`.
2. Solve the system `Ux = y`.

Solving the systems `Ly = b` and `Ux = y` is referred to as a forward solve and a backward solve, respectively.

If a symmetric matrix A is also positive definite, it can be shown that A can be factored as LL^T where L is a lower triangular matrix. Similarly, a Hermitian matrix, A, that is positive definite can be factored as $A = LL^H$. For both symmetric and Hermitian matrices, a factorization of this form is called a Cholesky factorization.

In a Cholesky factorization, the matrix U in an LU decomposition is either L^T or L^H. Consequently, a solver can increase its efficiency by only storing L, and one-half of A, and not computing U. Therefore, users who can express their application as the solution of a system of positive definite equations will gain a significant performance improvement over using a general representation.

For matrices that are symmetric (or Hermitian) but not positive definite, there are still some significant efficiencies to be had. It can be shown that if A is symmetric but not positive definite, then A can be factored as $A = LDL^T$, where D is a diagonal matrix and L is a lower unit triangular matrix. Similarly, if A is Hermitian, it can be factored as $A = LDL^H$. In either case, we again only need to store L, D, and half of A and we need not compute U. However, the backward solve phases must be amended to solving $L^T x = D^{-1} y$ rather than $L^T x = y$.

Fill-In and Reordering of Sparse Matrices

Two important concepts associated with the solution of sparse systems of equations are fill-in and reordering. The following example illustrates these concepts.

Consider the system of linear equation $Ax = b$, where A is the symmetric positive definite sparse matrix defined by the following:

$$A = \begin{bmatrix} 9 & \frac{3}{2} & 6 & \frac{3}{4} & 3 \\ \frac{3}{2} & \frac{1}{2} & * & * & * \\ 6 & * & 12 & * & * \\ \frac{3}{4} & * & * & \frac{5}{8} & * \\ 3 & * & * & * & 16 \end{bmatrix} \quad b = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

A star (*) is used to represent zeros and to emphasize the sparsity of A. The Cholesky factorization of A is: $A = LL^T$, where L is the following:

$$L = \begin{bmatrix} 3 & * & * & * & * \\ \frac{1}{2} & \frac{1}{2} & * & * & * \\ 2 & -2 & 2 & * & * \\ \frac{1}{4} & \frac{1}{-4} & \frac{1}{-2} & \frac{1}{2} & * \\ 1 & -1 & -2 & -3 & 1 \end{bmatrix}$$

Notice that even though the matrix *A* is relatively sparse, the lower triangular matrix *L* has no zeros below the diagonal. If we computed *L* and then used it for the forward and backward solve phase, we would do as much computation as if *A* had been dense.

The situation of *L* having non-zeros in places where *A* has zeros is referred to as fill-in. Computationally, it would be more efficient if a solver could exploit the non-zero structure of *A* in such a way as to reduce the fill-in when computing *L*. By doing this, the solver would only need to compute the non-zero entries in *L*. Toward this end, consider permuting the rows and columns of *A*. As described in Matrix Fundamentals section, the permutations of the rows of *A* can be represented as a permutation matrix, *P*. The result of permuting the rows is the product of *P* and *A*. Suppose, in the above example, we swap the first and fifth row of *A*, then swap the first and fifth columns of *A*, and call the resulting matrix *B*. Mathematically, we can express the process of permuting the rows and columns of *A* to get *B* as $B = PAP^T$. After permuting the rows and columns of *A*, we see that *B* is given by the following:

$$B = \begin{bmatrix} 16 & * & * & * & 3 \\ * & \frac{1}{2} & * & * & \frac{3}{2} \\ * & * & 12 & * & 6 \\ * & * & * & \frac{5}{8} & \frac{3}{4} \\ 3 & \frac{3}{2} & 6 & \frac{3}{4} & 9 \end{bmatrix}$$

Since *B* is obtained from *A* by simply switching rows and columns, the numbers of non-zero entries in *A* and *B* are the same. However, when we find the Cholesky factorization, $B = LL^T$, we see the following:

$$L = \begin{bmatrix} 4 & * & * & * & * \\ * & \frac{1}{\sqrt{2}} & * & * & * \\ * & * & 2(\sqrt{3}) & * & * \\ * & * & * & \frac{\sqrt{10}}{4} & * \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{\frac{3}{5}}}{4} \end{bmatrix}$$

The fill-in associated with *B* is much smaller than the fill-in associated with *A*. Consequently, the storage and computation time needed to factor *B* is much smaller than to factor *A*. Based on this, we see that an efficient sparse solver needs to find permutation *P* of the matrix *A*, which minimizes the fill-in for factoring $B = PAP^T$, and then use the factorization of *B* to solve the original system of equations.

Although the above example is based on a symmetric positive definite matrix and a Cholesky decomposition, the same approach works for a general *LU* decomposition. Specifically, let *P* be a permutation matrix, $B = PAP^T$ and suppose that *B* can be factored as $B = LU$. Then

$$
\begin{aligned}
& Ax = b \\
\Rightarrow\quad & PA(P^{-1}P)x = Pb \\
\Rightarrow\quad & PA(P^TP)x = Pb \\
\Rightarrow\quad & (PAP^T)(Px) = Pb \\
\Rightarrow\quad & B(Px) = Pb \\
\Rightarrow\quad & LU(Px) = Pb
\end{aligned}
$$

It follows that if we obtain an *LU* factorization for *B*, we can solve the original system of equations by a three step process:

1. Solve $Ly = Pb$.
2. Solve $Uz = y$.
3. Set $x = P^Tz$.

If we apply this three step process to the current example, we first need to perform the forward solve of the systems of equation $Ly = Pb$:

$$
\begin{bmatrix}
4 & * & * & * & * \\
* & \frac{1}{\sqrt{2}} & * & * & * \\
* & * & 2(\sqrt{3}) & * & * \\
* & * & * & \frac{\sqrt{10}}{4} & * \\
\frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{\frac{3}{5}}}{4}
\end{bmatrix}
*
\begin{bmatrix} y1 \\ y2 \\ y3 \\ y4 \\ y5 \end{bmatrix}
=
\begin{bmatrix} 5 \\ 2 \\ 3 \\ 4 \\ 1 \end{bmatrix}
$$

This gives: $y^T = \frac{5}{4}, 2\sqrt{2}, \frac{\sqrt{3}}{2}, \frac{16}{\sqrt{10}}, \frac{-979\sqrt{\frac{3}{5}}}{12}$.

The second step is to perform the backward solve, $Uz = y$. Or, in this case, since we are using a Cholesky factorization, $L^Tz = y$.

$$\begin{bmatrix} 4 & * & * & * & \frac{3}{4} \\ * & \frac{1}{\sqrt{2}} & * & * & \frac{3}{\sqrt{2}} \\ * & * & 2(\sqrt{3}) & * & \sqrt{3} \\ * & * & * & \frac{\sqrt{10}}{4} & \frac{3}{\sqrt{10}} \\ * & * & * & * & \frac{\sqrt{\frac{3}{5}}}{4} \end{bmatrix} * \begin{bmatrix} z1 \\ z2 \\ z3 \\ z4 \\ z5 \end{bmatrix} = \begin{bmatrix} \frac{5}{4} \\ 2(\sqrt{2}) \\ \frac{\sqrt{3}}{2} \\ \frac{16}{\sqrt{10}} \\ \frac{-979\sqrt{\frac{3}{5}}}{12} \end{bmatrix}$$

This gives $z = \frac{123}{2}, 983, \frac{1961}{12}, 398, \frac{-979}{3}$.

The third and final step is to set $x = P^T z$. This gives $x^T = \frac{-979}{3}, 983, \frac{1961}{12}, 398, \frac{123}{2}$.

Sparse Matrix Storage Format

As discussed above, it is more efficient to store only the non-zeros of a sparse matrix. This assumes that the sparsity is large, i.e., the number of non-zero entries is a small percentage of the total number of entries. If there is only an occasional zero entry, the cost of exploiting the sparsity actually slows down the computation when compared to simply treating the matrix as dense, meaning that all the values, zero and non-zero, are used in the computation.

There are a number of common storage schemes used for sparse matrices, but most of the schemes employ the same basic technique. That is, compress all of the non-zero elements of the matrix into a linear array, and then provide some number of auxiliary arrays to describe the locations of the non-zeros in the original matrix.

The compression of the non-zeros of a sparse matrix A into a linear array is done by walking down each column (column major format) or across each row (row major format) in order, and writing the non-zero elements to a linear array in the order that they appear in the walk.

When storing symmetric matrices, it is necessary to store only the upper triangular half of the matrix (upper triangular format) or the lower triangular half of the matrix (lower triangular format).

The Intel MKL direct sparse solver uses a row major upper triangular storage format. That is, the matrix is compressed row-by-row and for symmetric matrices only non-zeros in the upper triangular half of the matrix are stored.

The Intel MKL storage format for sparse matrices consists of three arrays, which are called the `values`, `columns`, and `rowIndex` arrays. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix *A*.

`values` — A real or complex array that contains the non-zero entries of *A*. The non-zero values of *A* are mapped into the `values` array using the row major, upper triangular storage mapping described above.

`columns` — Element `i` of the integer array `columns` contains the number of the column in *A* that contained the value in `values(i)`.

`rowIndex` — Element `j` of the integer array `rowIndex` gives the index into the `values` array that contains the first non-zero element in a row `j` of *A*. The length of the `values` and `columns` arrays is equal to the number of non-zeros in *A*.

Since the `rowIndex` array gives the location of the first non-zero within a row, and the non-zeros are stored consecutively, then we would like to be able to compute the number of non-zeros in the `i`-th row as the difference of `rowIndex(i)` and `rowIndex(i+1)`.

In order to have this relationship hold for the last row of *A*, we need to add an entry (dummy entry) to the end of `rowIndex` whose value is equal to the number of non-zeros in A, plus one. This makes the total length of the `rowIndex` array one larger than the number of rows of *A*.

NOTE. The Intel MKL sparse storage scheme uses the Fortran programming language convention of starting array indices at 1, rather than the C programming language convention of starting at 0.

With the above in mind, consider storing the symmetric matrix discussed in the example from the previous section.

$$A = \begin{bmatrix} 9 & \frac{3}{2} & 6 & \frac{3}{4} & 3 \\ * & \frac{1}{2} & * & * & * \\ * & * & \frac{1}{2} & * & * \\ * & * & * & \frac{5}{8} & * \\ * & * & * & * & 16 \end{bmatrix}$$

In this case, *A* has nine non-zero elements, so the lengths of the `values` and `columns` arrays will be nine. Also, since the matrix *A* has five rows, the `rowIndex` array is of length six. The actual values for each of the arrays for the example matrix are as follows:

Table 0-1 Storage Arrays for a Symmetric Example Matrix

values	=	(9	3/2	6	3/4	3	1/2	12	5/8	16)
columns	=	(1	2	3	4	5	2	3	4	5)
rowIndex	=	(1	6	7	8	9	10)			

For a non-symmetric or non-Hermitian array, all of the non-zeros need to be stored. Consider the non-symmetric matrix *B* defined by the following:

$$B = \begin{bmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & * & * & -5 \end{bmatrix}$$

We see that *B* has 13 non-zeros, and we store *B* as follows:

Table 0-2 Storage Arrays for a Non-Symmetric Example Matrix

values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
columns	=	(1	2	4	1	2	3	4	5	1	3	4	2	5)
rowIndex	=	(1	4	6	9	12	14)							

In the current version of Intel MKL, direct sparse solvers cannot solve non-symmetric systems of equations. However, it can solve symmetrically structured systems of equations.
A symmetrically structured system of equations is one where the pattern of non-zeros is symmetric. That is, a matrix has a symmetric structure if $a(j,i)$ is non-zero if and only if $a(j,i)$ is non-zero.

From the point of view of the solver software, a non-zero element of a matrix is anything that is stored in the `values` array. In that sense, we can turn any non-symmetric matrix into a symmetrically structured matrix by carefully adding zeros to the `values` array.
For example, suppose we consider the matrix *B* to have the following set of non-zero entries:

$$B = \begin{bmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & 0 \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & 0 & * & -5 \end{bmatrix}$$

Now *B* can be considered to be symmetrically structured with 15 non-zero entries.We would represent the matrix as:

Table 0-3 Storage Arrays for a Symmetrically Structured Example Matrix

`values`	=	(1	-1	-3	-2	5	0	4	6	4	-4	2	7	8	0	-5)
`columns`	=	(1	2	4	1	2	5	3	4	5	1	3	4	2	3	5)
`rowIndex`	=	(1	4	7	10	13	16)									

Storage Format Restrictions

The storage format for the sparse solver must conform to two important restrictions:

First, the non-zero values in a given row must be placed into the `values` array in the order in which they occur in the row (from left to right). Second, no diagonal element can be omitted from the `values` array for any symmetric or structurally symmetric matrix.

The second restriction implies that when dealing with symmetric or structurally symmetric matrices that have zeros on the diagonal, the zero diagonal elements must be explicitly represented in the `values` array.

Routine and Function Arguments

B

The major arguments in the BLAS routines are vector and matrix, whereas VML functions work on vector arguments only.
The sections that follow discuss each of these arguments and provide examples.

Vector Arguments in BLAS

Vector arguments are passed in one-dimensional arrays. The array dimension (length) and vector increment are passed as integer variables. The length determines the number of elements in the vector. The increment (also called stride) determines the spacing between vector elements and the order of the elements in the array in which the vector is passed.

A vector of length `n` and increment `incx` is passed in a one-dimensional array `x` whose values are defined as

`x(1), x(1+|incx|), ..., x(1+(n-1)* |incx|)`

If `incx` is positive, then the elements in array `x` are stored in increasing order. If `incx` is negative, the elements in array `x` are stored in decreasing order with the first element defined as `x(1+(n-1)* |incx|)`. If `incx` is zero, then all elements of the vector have the same value, `x(1)`. The dimension of the one-dimensional array that stores the vector must always be at least

`idimx = 1 + (n-1)* |incx |`

Example B-1 One-dimensional Real Array

Let `x(1:7)` be the one-dimensional real array

`x = (1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0)`.

If *incx* =2 and *n* = 3, then the vector argument with elements in order from first to last is `(1.0, 5.0, 9.0)`.

If *incx* = -2 and *n* = 4, then the vector elements in order from first to last is `(13.0, 9.0, 5.0, 1.0)`.

If *incx* = 0 and *n* = 4, then the vector elements in order from first to last is `(1.0, 1.0, 1.0, 1.0)`.

One-dimensional substructures of a matrix, such as the rows, columns, and diagonals, can be passed as vector arguments with the starting address and increment specified. In Fortran, storing the *m* by *n* matrix is based on column-major ordering where the increment between elements in the same column is `1`, the increment between elements in the same row is *m*, and the increment between elements on the same diagonal is *m* + 1.

Example B-2 Two-dimensional Real Matrix

Let *a* be the real 5 x 4 matrix declared as `REAL A (5,4)`.

To scale the third column of *a* by 2.0, use the BLAS routine `sscal` with the following calling sequence:

```
call sscal (5, 2.0, a(1,3), 1).
```

To scale the second row, use the statement:

```
call sscal (4, 2.0, a(2,1), 5).
```

To scale the main diagonal of A by 2.0, use the statement:

```
call sscal (5, 2.0, a(1,1), 6).
```

NOTE. The default vector argument is assumed to be 1.

Vector Arguments in VML

Vector arguments of VML mathematical functions are passed in one-dimensional arrays with unit vector increment. It means that a vector of length *n* is passed contiguously in an array a whose values are defined as `a[0], a[1], ..., a[n-1]` (for C- interface).

To accommodate for arrays with other increments, or more complicated indexing, VML contains auxiliary pack/unpack functions that gather the array elements into a contiguous vector and then scatter them after the computation is complete.

Generally, if the vector elements are stored in a one-dimensional array `a` as

```
a[m0], a[m1], ..., a[mn-1]
```

and need to be regrouped into an array *y* as

```
y[k0], y[k1], ..., y[kn-1],
```

VML pack/unpack functions can use one of the following indexing methods:

Positive Increment Indexing

```
kj = incy * j, mj = inca * j,       j = 0 ,…, n-1
```

Constraint: `incy > 0` and `inca > 0`.
For example, setting `incy = 1` specifies gathering array elements into a contiguous vector.

This method is similar to that used in BLAS, with the exception that negative and zero increments are not permitted.

Index Vector Indexing

```
kj = iy[j], mj = ia[j],   j = 0 ,…, n-1,
```

where `ia` and `iy` are arrays of length *n* that contain index vectors for the input and output arrays `a` and `y`, respectively.

Mask Vector Indexing

Indices `kj, mj` are such that:

```
my[kj] ≠ 0, ma[mj] ≠ 0 ,   j = 0,…, n-1,
```

where `ma` and `my` are arrays that contain mask vectors for the input and output arrays `a` and `y`, respectively.

Matrix Arguments

Matrix arguments of the Intel® Math Kernel Library routines can be stored in either one- or two-dimensional arrays, using the following storage schemes:

- conventional full storage (in a two-dimensional array)
- packed storage for Hermitian, symmetric, or triangular matrices (in a one-dimensional array)
- band storage for band matrices (in a two-dimensional array).

Full storage is the following obvious scheme: a matrix A is stored in a two-dimensional array `a`, with the matrix element a_{ij} stored in the array element `a(i,j)`.

If a matrix is *triangular* (upper or lower, as specified by the argument `uplo`), only the elements of the relevant triangle are stored; the remaining elements of the array need not be set.

Routines that handle symmetric or Hermitian matrices allow for either the upper or lower triangle of the matrix to be stored in the corresponding elements of the array:

if `uplo ='U'`, a_{ij} is stored in `a(i,j)` for $i \le j$, other elements of `a` need not be set.

if `uplo ='L'`, a_{ij} is stored in `a(i,j)` for $j \le i$, other elements of `a` need not be set.

Packed storage allows you to store symmetric, Hermitian, or triangular matrices more compactly: the relevant triangle (again, as specified by the argument `uplo`) is packed by columns in a one-dimensional array `ap`:

if `uplo ='U'`, a_{ij} is stored in `ap(i+j(j-1)/2)` for $i \le j$

if `uplo ='L'`, a_{ij} is stored in `ap(i+(2*n-j)*(j-1)/2)` for $j \le i$.

In descriptions of LAPACK routines, arrays with packed matrices have names ending in `p`.

Band storage is as follows: an `m` by `n` band matrix with `kl` non-zero sub-diagonals and `ku` non-zero super-diagonals is stored compactly in a two-dimensional array `ab` with `kl`+`ku`+1 rows and `n` columns. Columns of the matrix are stored in the corresponding columns of the array, and diagonals of the matrix are stored in rows of the array. Thus,

a_{ij} is stored in `ab(kl+ku+1+i-j,j)` for max(`n`,`j-ku`) $\le$ `i` $\le$ min(`n`,`j`+`kl`).

Use the band storage scheme only when `kl` and `ku` are much less than the matrix size `n`. (Although the routines work correctly for all values of `kl` and `ku`, it's inefficient to use the band storage if your matrices are not really banded).

When a general band matrix is supplied for *LU factorization*, space must be allowed to store `kl` additional super-diagonals generated by fill-in as a result of row interchanges. This means that the matrix is stored according to the above scheme, but with `kl` + `ku` super-diagonals.

The band storage scheme is illustrated by the following example, when
`m = n = 6, kl = 2, ku = 1`:

Banded matrix A

$$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & 0 & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} \end{bmatrix}$$

Band storage of A

$$\begin{matrix} * & * & * & + & + & + \\ * & * & + & + & + & + \\ * & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & * \\ a_{31} & a_{42} & a_{53} & a_{64} & * & * \end{matrix}$$

Array elements marked * are not used by the routines; elements marked + need not be set on entry, but are required by the LU factorization routines to store the results. The input array will be overwritten on exit by the details of the LU factorization as follows:

$$\begin{matrix} * & * & * & u_{14} & u_{25} & u_{36} \\ * & * & u_{13} & u_{24} & u_{35} & u_{46} \\ * & u_{12} & u_{23} & u_{34} & u_{45} & u_{56} \\ u_{11} & u_{22} & u_{33} & u_{44} & u_{55} & u_{66} \\ m_{21} & m_{32} & m_{43} & m_{54} & m_{65} & * \\ m_{31} & m_{42} & m_{53} & m_{64} & * & * \end{matrix}$$

where u_{ij} are the elements of the upper triangular matrix U, and m_{ij} are the multipliers used during factorization.

Triangular band matrices are stored in the same format, with either $kl = 0$ if upper triangular, or $ku = 0$ if lower triangular. For symmetric or Hermitian band matrices with k sub-diagonals or super-diagonals, you need to store only the upper or lower triangle, as specified by the argument *uplo*:

if *uplo* = `'U'`, a_{ij} is stored in $ab(k+1+i-j, j)$ for $\max(1, j-k) \le i \le j$
if *uplo* = `'L'`, a_{ij} is stored in $ab(1+i-j, j)$ for $j \le i \le \min(n, j+k)$.

In descriptions of LAPACK routines, arrays that hold matrices in band storage have names ending in *b*.

In Fortran, column-major ordering of storage is assumed. This means that elements of the same column occupy successive storage locations.

Three quantities are usually associated with a two-dimensional array argument: its leading dimension, which specifies the number of storage locations between elements in the same row, its number of rows, and its number of columns. For a matrix in full storage, the leading dimension of the array must be at least as large as the number of rows in the matrix.

A character transposition parameter is often passed to indicate whether the matrix argument is to be used in normal or transposed form or, for a complex matrix, if the conjugate transpose of the matrix is to be used.
The values of the transposition parameter for these three cases are the following:

`'N'` or `'n'`	normal (no conjugation, no transposition)
`'T'` or `'t'`	transpose
`'C'` or `'c'`	conjugate transpose.

Example B-3 Two-Dimensional Complex Array

Suppose `A (1:5, 1:4)` is the complex two-dimensional array presented by matrix

$$\begin{bmatrix} (1.1, 0.11) & (1.2, 0.12) & (1.3, 0.13) & (1.4, 0.14) \\ (2.1, 0.21) & (2.2, 0.22) & (2.3, 0.23) & (2.4, 0.24) \\ (3.1, 0.31) & (3.2, 0.32) & (3.3, 0.33) & (3.4, 0.34) \\ (4.1, 0.41) & (4.2, 0.42) & (4.3, 0.43) & (4.4, 0.44) \\ (5.1, 0.51) & (5.2, 0.52) & (5.3, 0.53) & (5.4, 0.54) \end{bmatrix}$$

Let *transa* be the transposition parameter, *m* be the number of rows, *n* be the number of columns, and *lda* be the leading dimension. Then if `transa = 'N', m = 4, n = 2`, and `lda = 5`, the matrix argument would be

$$\begin{bmatrix} (1.1, 0.11) & (1.2, 0.12) \\ (2.1, 0.21) & (2.2, 0.22) \\ (3.1, 0.31) & (3.2, 0.32) \\ (4.1, 0.41) & (4.2, 0.42) \end{bmatrix}$$

If `transa = 'T', m = 4, n = 2`, and `lda =5`,
the matrix argument would be

$$\begin{bmatrix} (1.1, 0.11) & (2.1, 0.21) & (3.1, 0.31) & (4.1, 0.41) \\ (1.2, 0.12) & (2.2, 0.22) & (3.2, 0.32) & (4.2, 0.42) \end{bmatrix}$$

If `transa = 'C', m = 4, n = 2`, and `lda =5`,
the matrix argument would be

$$\begin{bmatrix} (1.1, -0.11) & (2.1, -0.21) & (3.1, -0.31) & (4.1, -0.41) \\ (1.2, -0.12) & (2.2, -0.22) & (3.2, -0.32) & (4.2, -0.42) \end{bmatrix}$$

Note that care should be taken when using a leading dimension value which is different from the number of rows specified in the declaration of the two-dimensional array. For example, suppose the array *A* above is declared as `COMPLEX A (5,4)`.

continued <TableFinger>*

Then if `transa = 'N', m = 3, n = 4`, and `lda = 4`, the matrix argument will be

$$\begin{bmatrix} (1.1, 0.11) & (5.1, 0.51) & (4.2, 0.42) & (3.3, 0.33) \\ (2.1, 0.21) & (1.2, 0.12) & (5.2, 0.52) & (4.3, 0.43) \\ (3.1, 0.31) & (2.2, 0.22) & (1.3, 0.13) & (5.3, 0.53) \end{bmatrix}$$

Code Examples

C

This appendix presents code examples of using some Intel MKL routines and functions. You can find here example code written in both Fortran and C.

Currently, the appendix includes the following sections:

- BLAS Code Examples
- PARDISO Code Examples
- Direct Sparse Solver Examples
- DFT Code Examples

Please refer to respective chapters in the manual for detailed descriptions of function parameters and operation.

BLAS Code Examples

Example C-1 Using BLAS Level 1 Function

The following example illustrates a call to the BLAS Level 1 function `sdot`. This function performs a vector-vector operation of computing a scalar product of two single-precision real vectors *x* and *y*.

Parameters

n Specifies the order of vectors *x* and *y*.

incx Specifies the increment for the elements of *x*.

incy Specifies the increment for the elements of *y*.

```
program dot_main
real x(10), y(10), sdot, res
integer n, incx, incy, i
external sdot

n = 5
incx = 2
incy = 1

do i = 1, 10
  x(i) = 2.0e0
  y(i) = 1.0e0
end do

res = sdot (n, x, incx, y, incy)

print*, 'SDOT = ', res

end
```

As a result of this program execution, the following line is printed:

SDOT = 10.000

Example C-2 Using BLAS Level 1 Routine

The following example illustrates a call to the BLAS Level 1 routine `scopy`. This routine performs a vector-vector operation of copying a single-precision real vector *x* to a vector *y*.

Parameters

n Specifies the order of vectors *x* and *y*.

incx Specifies the increment for the elements of *x*.

incy Specifies the increment for the elements of *y*.

```
program copy_main
real x(10), y(10)
integer n, incx, incy, i

n = 3

incx = 3

incy = 1

do i = 1, 10

  x(i) = i
```

```
end do
call scopy (n, x, incx, y, incy)
print*, 'Y = ', (y(i), i = 1, n)
end
```

As a result of this program execution, the following line is printed:

Y = 1.00000 4.00000 7.00000

Example C-3 Using BLAS Level 2 Routine

The following example illustrates a call to the BLAS Level 2 routine `sger`. This routine performs a matrix-vector operation

a := *alpha***x***y*' + *a*.

Parameters

alpha Specifies a scalar *alpha*.

x *m*-element vector.

y *n*-element vector.

a *m* by *n* matrix.

```
program ger_main
real a(5,3), x(10), y(10), alpha
integer m, n, incx, incy, i, j, lda
m = 2
n = 3
lda = 5
incx = 2
incy = 1
alpha = 0.5
do i = 1, 10
  x(i) = 1.0
  y(i) = 1.0
end do
```

```
do i = 1, m
  do j = 1, n
    a(i,j) = j
  end do
end do
call sger (m, n, alpha, x, incx, y, incy, a, lda)
print*, 'Matrix A: '
do i = 1, m
  print*, (a(i,j), j = 1, n)
end do
end
```

As a result of this program execution, matrix *a* is printed as follows:

Matrix A:

1.50000 2.50000 3.50000

1.50000 2.50000 3.50000

Example C-4 Using BLAS Level 3 Routine

The following example illustrates a call to the BLAS Level 3 routine `ssymm`. This routine performs a matrix-matrix operation

*c := alpha*a*b' + beta*c*.

Parameters

alpha	Specifies a scalar *alpha*.
beta	Specifies a scalar *beta*.
a	Symmetric matrix.
b	*m* by *n* matrix.
c	*m* by *n* matrix.

```
program symm_main
real a(3,3), b(3,2), c(3,3), alpha, beta
integer m, n, lda, ldb, ldc, i, j
```

```
character uplo, side
uplo = 'u'
side = 'l'
m = 3
n = 2
lda = 3
ldb = 3
ldc = 3
alpha = 0.5
beta = 2.0
do i = 1, m
  do j = 1, m
    a(i,j) = 1.0
  end do
end do
do i = 1, m
  do j = 1, n
    c(i,j) = 1.0
    b(i,j) = 2.0
  end do
end do
call ssymm (side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
print*, ‘Matrix C: ‘
do i = 1, m
  print*, (c(i,j), j = 1, n)
end do
end
```

As a result of this program execution, matrix *c* is printed as follows:

Matrix C:

5.00000 5.00000

5.00000 5.00000

5.00000 5.00000

Example C-5 Calling a Complex BLAS Level 1 Function from C

The following example illustrates a call from a C program to the complex BLAS Level 1 function `zdotc()`. This function computes the dot product of two double-precision complex vectors.

In this example, the complex dot product is returned in the structure `c`.

```
#define N 5
void main()
{

  int n, inca = 1, incb = 1, i;
  typedef struct{ double re; double im; } complex16;
  complex16 a[N], b[N], c;
  void zdotc();
  n = N;
  for( i = 0; i < n; i++ ){
    a[i].re = (double)i; a[i].im = (double)i * 2.0;
    b[i].re = (double)(n - i); b[i].im = (double)i * 2.0;
  }
  zdotc( &c, &n, a, &inca, b, &incb );
  printf( "The complex dot product is: ( %6.2f, %6.2f )\n", c.re, c.im );
}
```

NOTE. Instead of calling BLAS directly from C programs, you might wish to use the CBLAS interface; this is the supported way of calling BLAS from C. For more information about CBLAS, see Appendix D, which presents CBLAS, the C interface to the Basic Linear Algebra Subprograms (BLAS) implemented in Intel® MKL..

PARDISO Code Examples

This section presents code examples of using the PARDISO direct solver for computing solutions of linear systems with sparse matrices. For description of this solver, refer to Chapter 8 of the manual.

Examples for sparse symmetric linear systems

In this section two examples (Fortran, C) are provided to solve symmetric linear systems with PARDISO. To solve the systems of equations $Ax = b$, where

$$A = \begin{bmatrix} 7.0 & 0.0 & 1.0 & 0.0 & 0.0 & 2.0 & 7.0 & 0.0 \\ 0.0 & -4.0 & 8.0 & 0.0 & 2.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 8.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 5.0 \\ 0.0 & 0.0 & 0.0 & 7.0 & 0.0 & 0.0 & 9.0 & 0.0 \\ 0.0 & 2.0 & 0.0 & 0.0 & 5.0 & 1.0 & 5.0 & 0.0 \\ 2.0 & 0.0 & 0.0 & 0.0 & 1.0 & -1.0 & 0.0 & 5.0 \\ 7.0 & 0.0 & 0.0 & 9.0 & 5.0 & 0.0 & 11.0 & 0.0 \\ 0.0 & 0.0 & 5.0 & 0.0 & 0.0 & 5.0 & 0.0 & 5.0 \end{bmatrix} \quad \text{and} \quad \mathrm{B} = \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{bmatrix}$$

Example results for symmetric systems

Upon successful execution of the solver, the result of the solution X is as follows

```
Reordering completed ...
Number of nonzeros in factors = 30
Number of factorization MFLOPS = 0
Factorization completed ...
Solve completed ...
The solution of the system is
x(1) = -0.0418602013
x(2) = -0.00341312416
x(3) = 0.117250377
x(4) = -0.11263958
x(5) = 0.0241722445
```

```
        x(6) = -0.10763334
        x(7) = 0.198719673
        x(8) = 0.190382964
```

Example C-6 Example pardiso_sym.f for symmetric linear systems

```
C-----------------------------------------------------------------------
C Example program to show the use of the "PARDISO" routine
C for symmetric linear systems
C----------------------------------------------------------------------
C This program can be downloaded from the following site:
C http://www.computational.unibas.ch/cs/scicomp
C
C (C) Olaf Schenk, Department of Computer Science,
C University of Basel, Switzerland.
C Email: olaf.schenk@unibas.ch
C
C----------------------------------------------------------------------
      PROGRAM pardiso_sym
      IMPLICIT NONE
C.. Internal solver memory pointer for 64-bit architectures
C.. INTEGER*8 pt(64)
C.. Internal solver memory pointer for 32-bit architectures
C.. INTEGER*4 pt(64)
C.. This is OK in both cases
      INTEGER*8 pt(64)
C.. All other variables
      INTEGER maxfct, mnum, mtype, phase, n, nrhs, error, msglvl
      INTEGER iparm(64)
      INTEGER ia(9)
      INTEGER ja(18)
      REAL*8 a(18)
      REAL*8 b(8)
      REAL*8 x(8)
```

```
      INTEGER i, idum
      REAL*8 waltime1, waltime2, ddum
C.. Fill all arrays containing matrix data.
      DATA n /8/, nrhs /1/, maxfct /1/, mnum /1/
      DATA ia /1,5,8,10,12,15,17,18,19/
      DATA ja
     1 /1,  3,    6,7,
     2    2,3,  5,
     3      3,        8,
     4        4,    7,
     5          5,6,7,
     6            6,  8,
     7              7,
     8                8/
      DATA a
     1 /7.d0,     1.d0,          2.d0,7.d0,
     2       -4.d0,8.d0,     2.d0,
     3            1.d0,                    5.d0,
     4                 7.d0,     9.d0,
     5                      5.d0,1.d0,5.d0,
     6                           -1.d0,     5.d0,
     7                                11.d0,
     8                                     5.d0/
      integer omp_get_max_threads
      external omp_get_max_threads
C..
C.. Set up PARDISO control parameter
C..
      do i = 1, 64
         iparm(i) = 0
      end do
      iparm(1) = 1 ! no solver default
      iparm(2) = 2 ! fill-in reordering from METIS
```

```
      iparm(3) = omp_get_max_threads() !numbers of processors, value of OMP_NUM_THREADS
      iparm(4) = 0 ! no iterative-direct algorithm
      iparm(5) = 0 ! no user fill-in reducing permutation
      iparm(6) = 0 ! =0 solution on the first n compoments of x
      iparm(7) = 16 ! default logical fortran unit number for output
      iparm(8) = 9 ! numbers of iterative refinement steps
      iparm(9) = 0 ! not in use
      iparm(10) = 13 ! perturbe the pivot elements with 1E-13
      iparm(11) = 1 ! use nonsymmetric permutation and scaling MPS
      iparm(12) = 0 ! not in use
      iparm(13) = 0 ! not in use
      iparm(14) = 0 ! Output: number of perturbed pivots
      iparm(15) = 0 ! not in use
      iparm(16) = 0 ! not in use
      iparm(17) = 0 ! not in use
      iparm(18) = -1 ! Output: number of nonzeros in the factor LU
      iparm(19) = -1 ! Output: Mflops for LU factorization
      iparm(20) = 0 ! Output: Numbers of CG Iterations
      error = 0 ! initialize error flag
      msglvl = 0 ! don't print statistical information
      mtype = -2 ! unsymmetric matrix symmetric, indefinite, no pivoting
C.. Initiliaze the internal solver memory pointer. This is only
C necessary for the FIRST call of the PARDISO solver.
      do i = 1, 64
         pt(i) = 0
      end do
C.. Reordering and Symbolic Factorization, This step also allocates
C all memory that is necessary for the factorization
      phase = 11 ! only reordering and symbolic factorization
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
     1 idum, nrhs, iparm, msglvl, ddum, ddum, error)
      WRITE(*,*) 'Reordering completed ... '
      IF (error .NE. 0) THEN
```

```
         WRITE(*,*) 'The following ERROR was detected: ', error
         STOP
      END IF
      WRITE(*,*) 'Number of nonzeros in factors = ',iparm(18)
      WRITE(*,*) 'Number of factorization MFLOPS = ',iparm(19)
C.. Factorization.
      phase = 22 ! only factorization
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
     1 idum, nrhs, iparm, msglvl, ddum, ddum, error)
      WRITE(*,*) 'Factorization completed ... '
      IF (error .NE. 0) THEN
         WRITE(*,*) 'The following ERROR was detected: ', error
         STOP
      ENDIF
C.. Back substitution and iterative refinement
      iparm(8) = 2 ! max numbers of iterative refinement steps
      phase = 33 ! only factorization
      do i = 1, n
         b(i) = 1.d0
      end do
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
     1 idum, nrhs, iparm, msglvl, b, x, error)
      WRITE(*,*) 'Solve completed ... '
      WRITE(*,*) 'The solution of the system is '
      DO i = 1, n
         WRITE(*,*) ' x(',i,') = ', x(i)
      END DO
C.. Termination and release of memory
      phase = -1 ! release internal memory
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, ddum, idum, idum,
     1 idum, nrhs, iparm, msglvl, ddum, ddum, error)
      END
```

Example C-7 Example pardiso_sym.c for symmetric linear systems

```
/* -------------------------------------------------------------------- */
/* Example program to show the use of the "PARDISO" routine */
/* on symmetric linear systems */
/* -------------------------------------------------------------------- */
/* This program can be downloaded from the following site: */
/* http://www.computational.unibas.ch/cs/scicomp */
/* */
/* (C) Olaf Schenk, Department of Computer Science, */
/* University of Basel, Switzerland. */
/* Email: olaf.schenk@unibas.ch */
/* -------------------------------------------------------------------- */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
extern int omp_get_max_threads();
/* PARDISO prototype. */
extern int PARDISO
      (void *, int *, int *, int *, int *, int *,
      double *, int *, int *, int *, int *, int *,
      int *, double *, double *, int *);

int main( void ) {
      /* Matrix data. */
      int n = 8;
      int ia[ 9] = { 1, 5, 8, 10, 12, 15, 17, 18, 19 };
      int ja[18] = { 1, 3, 6, 7,
            2, 3, 5,
            3, 8,
            4, 7,
            5, 6, 7,
            6, 8,
```

```
            7,
            8 };
    double a[18] = { 7.0, 1.0, 2.0, 7.0,
            -4.0, 8.0, 2.0,
            1.0, 5.0,
            7.0, 9.0,
            5.0, 1.0, 5.0,
            -1.0, 5.0,
            11.0,
            5.0 };
    int mtype = -2; /* Real symmetric matrix */
    /* RHS and solution vectors. */
    double b[8], x[8];
    int nrhs = 1; /* Number of right hand sides. */
    /* Internal solver memory pointer pt, */
    /* 32-bit: int pt[64]; 64-bit: long int pt[64] */
    /* or void *pt[64] should be OK on both architectures */
    void *pt[64];
    /* Pardiso control parameters. */
    int iparm[64];
    int maxfct, mnum, phase, error, msglvl;
    /* Auxiliary variables. */
    int i;
    double ddum; /* Double dummy */
    int idum; /* Integer dummy. */
/* -------------------------------------------------------------------- */
/* .. Setup Pardiso control parameters. */
/* -------------------------------------------------------------------- */
    for (i = 0; i < 64; i++) {
            iparm[i] = 0;
    }
    iparm[0] = 1; /* No solver default */
    iparm[1] = 2; /* Fill-in reordering from METIS */
```

```
      /* Numbers of processors, value of OMP_NUM_THREADS */
      iparm[2] = omp_get_max_threads();
      iparm[3] = 0; /* No iterative-direct algorithm */
      iparm[4] = 0; /* No user fill-in reducing permutation */
      iparm[5] = 0; /* Write solution into x */
      iparm[6] = 16; /* Default logical fortran unit number for output */
      iparm[7] = 2; /* Max numbers of iterative refinement steps */
      iparm[8] = 0; /* Not in use */
      iparm[9] = 13; /* Perturb the pivot elements with 1E-13 */
      iparm[10] = 1; /* Use nonsymmetric permutation and scaling MPS */
      iparm[11] = 0; /* Not in use */
      iparm[12] = 0; /* Not in use */
      iparm[13] = 0; /* Output: Number of perturbed pivots */
      iparm[14] = 0; /* Not in use */
      iparm[15] = 0; /* Not in use */
      iparm[16] = 0; /* Not in use */
      iparm[17] = -1; /* Output: Number of nonzeros in the factor LU */
      iparm[18] = -1; /* Output: Mflops for LU factorization */
      iparm[19] = 0; /* Output: Numbers of CG Iterations */
      maxfct = 1; /* Maximum number of numerical factorizations. */
      mnum = 1; /* Which factorization to use. */
      msglvl = 0; /* Don't print statistical information in file */
      error = 0; /* Initialize error flag */
/* -------------------------------------------------------------------- */
/* .. Initialize the internal solver memory pointer. This is only */
/* necessary for the FIRST call of the PARDISO solver. */
/* -------------------------------------------------------------------- */
      for (i = 0; i < 64; i++) {
            pt[i] = 0;
      }
/* -------------------------------------------------------------------- */
/* .. Reordering and Symbolic Factorization. This step also allocates */
/* all memory that is necessary for the factorization. */
```

```
/* -------------------------------------------------------------------- */
      phase = 11;
      PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
            &n, a, ia, ja, &idum, &nrhs,
            iparm, &msglvl, &ddum, &ddum, &error);
      if (error != 0) {
            printf("\nERROR during symbolic factorization: %d", error);
            exit(1);
      }
      printf("\nReordering completed ... ");
      printf("\nNumber of nonzeros in factors = %d", iparm[17]);
      printf("\nNumber of factorization MFLOPS = %d", iparm[18]);
/* -------------------------------------------------------------------- */
/* .. Numerical factorization. */
/* -------------------------------------------------------------------- */
      phase = 22;
      PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
            &n, a, ia, ja, &idum, &nrhs,
            iparm, &msglvl, &ddum, &ddum, &error);
      if (error != 0) {
            printf("\nERROR during numerical factorization: %d", error);
            exit(2);
      }
      printf("\nFactorization completed ... ");
/* -------------------------------------------------------------------- */
/* .. Back substitution and iterative refinement. */
/* -------------------------------------------------------------------- */
      phase = 33;
      iparm[7] = 2; /* Max numbers of iterative refinement steps. */
      /* Set right hand side to one. */
      for (i = 0; i < n; i++) {
            b[i] = 1;
      }
```

```
      PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
            &n, a, ia, ja, &idum, &nrhs,
            iparm, &msglvl, b, x, &error);
      if (error != 0) {
            printf("\nERROR during solution: %d", error);
            exit(3);
      }
      printf("\nSolve completed ... ");
      printf("\nThe solution of the system is: ");
      for (i = 0; i < n; i++) {
            printf("\n x [%d] = % f", i, x[i] );
      }
      printf ("\n");
/* -------------------------------------------------------------------- */
/* .. Termination and release of memory. */
/* -------------------------------------------------------------------- */
      phase = -1; /* Release internal memory. */
      PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
            &n, &ddum, ia, ja, &idum, &nrhs,
            iparm, &msglvl, &ddum, &ddum, &error);
      return 0;
}
```

Examples for sparse unsymmetric linear systems

In this section two examples (Fortran, C) are provided to solve unsymmetric linear systems with PARDISO. To solve the systems of equations $Ax = b$, where

$$A = \begin{bmatrix} 1.0 & -1.0 & 0.0 & -3.0 & 0.0 \\ -2.0 & 5.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 4.0 & 6.0 & 4.0 \\ -4.0 & 0.0 & 2.0 & 7.0 & 0.0 \\ 0.0 & 8.0 & 0.0 & 0.0 & -5.0 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{bmatrix}$$

Example results for unsymmetric systems

Upon successful execution of the solver, the result of the solution X is as follows

```
Reordering completed ...
 Number of nonzeros in factors =  21
 Number of factorization MFLOPS =  0
 Factorization completed ...
 Solve completed ...
 The solution of the system is
  x( 1) =  -0.522321429
  x( 2) =  -0.00892857143
  x( 3) =   1.22098214
  x( 4) =  -0.504464286
  x( 5) =  -0.214285714
```

Example C-8 Example pardiso_unsym.f for unsymmetric linear systems

```
*******************************************************************************
*                               INTEL CONFIDENTIAL
*    Copyright(C) 2004 Intel Corporation. All Rights Reserved.
*    The source code contained  or  described herein and all documents related to
*    the source code ("Material") are owned by Intel Corporation or its suppliers
*    or licensors.  Title to the  Material remains with  Intel Corporation or its
*    suppliers and licensors. The Material contains trade secrets and proprietary
```

```
*   and  confidential  information of  Intel or its suppliers and licensors. The
*   Material  is  protected  by  worldwide  copyright  and trade secret laws and
*   treaty  provisions. No part of the Material may be used, copied, reproduced,
*   modified, published, uploaded, posted, transmitted, distributed or disclosed
*   in any way without Intel's prior express written permission.
*   No license  under any  patent, copyright, trade secret or other intellectual
*   property right is granted to or conferred upon you by disclosure or delivery
*   of the Materials,  either expressly, by implication, inducement, estoppel or
*   otherwise.  Any  license  under  such  intellectual property  rights must be
*   express and approved by Intel in writing.
*
********************************************************************************
*
*   Content : MKL DSS Fortran-77 example
*
********************************************************************************
*
C----------------------------------------------------------------------
C Example program to show the use of the "PARDISO" routine
C for symmetric linear systems
C---------------------------------------------------------------------
C This program can be downloaded from the following site:
C http://www.computational.unibas.ch/cs/scicomp
C
C (C) Olaf Schenk, Department of Computer Science,
C University of Basel, Switzerland.
C Email: olaf.schenk@unibas.ch
C
C---------------------------------------------------------------------
      PROGRAM pardiso_unsym
      IMPLICIT NONE
C.. Internal solver memory pointer for 64-bit architectures
C.. INTEGER*8 pt(64)
```

```
C.. Internal solver memory pointer for 32-bit architectures
C.. INTEGER*4 pt(64)
C.. This is OK in both cases
      INTEGER*8 pt(64)
C.. All other variables
      INTEGER maxfct, mnum, mtype, phase, n, nrhs, error, msglvl
      INTEGER iparm(64)
      INTEGER ia(6)
      INTEGER ja(13)
      REAL*8 a(13)
      REAL*8 b(5)
      REAL*8 x(5)
      INTEGER i, idum
      REAL*8 waltime1, waltime2, ddum
C.. Fill all arrays containing matrix data.
      DATA n /5/, nrhs /1/, maxfct /1/, mnum /1/
      DATA ia /1,4,6,9,12,14/
      DATA ja
     1 /   1,    2,          4,
     2     1,    2,
     3                 3,    4,    5,
     4     1,          3,    4,
     5           2,                5/
      DATA a
     1 /1.d0,-1.d0,      -3.d0,
     2 -2.d0, 5.d0,
     3              4.d0, 6.d0, 4.d0,
     4 -4.d0,       2.d0, 7.d0,
     5        8.d0,            -5.d0/
      integer omp_get_max_threads
      external omp_get_max_threads
C..
C.. Set up PARDISO control parameter
```

```
C..
      do i = 1, 64
         iparm(i) = 0
      end do
      iparm(1) = 1 ! no solver default
      iparm(2) = 2 ! fill-in reordering from METIS
      iparm(3) = omp_get_max_threads() ! numbers of processors, value of
OMP_NUM_THREADS
      iparm(4) = 0 ! no iterative-direct algorithm
      iparm(5) = 0 ! no user fill-in reducing permutation
      iparm(6) = 0 ! =0 solution on the first n compoments of x
      iparm(7) = 0 ! not in use
      iparm(8) = 9 ! numbers of iterative refinement steps
      iparm(9) = 0 ! not in use
      iparm(10) = 13 ! perturbe the pivot elements with 1E-13
      iparm(11) = 1 ! use nonsymmetric permutation and scaling MPS
      iparm(12) = 0 ! not in use
      iparm(13) = 0 ! not in use
      iparm(14) = 0 ! Output: number of perturbed pivots
      iparm(15) = 0 ! not in use
      iparm(16) = 0 ! not in use
      iparm(17) = 0 ! not in use
      iparm(18) = -1 ! Output: number of nonzeros in the factor LU
      iparm(19) = -1 ! Output: Mflops for LU factorization
      iparm(20) = 0 ! Output: Numbers of CG Iterations
      error = 0 ! initialize error flag
      msglvl = 1 ! print statistical information
      mtype = 11 ! real unsymmetric
C.. Initiliaze the internal solver memory pointer. This is only
C necessary for the FIRST call of the PARDISO solver.
      do i = 1, 64
         pt(i) = 0
      end do
```

```
C.. Reordering and Symbolic Factorization, This step also allocates
C all memory that is necessary for the factorization
      phase = 11 ! only reordering and symbolic factorization
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
     1 idum, nrhs, iparm, msglvl, ddum, ddum, error)
      WRITE(*,*) 'Reordering completed ... '
      IF (error .NE. 0) THEN
         WRITE(*,*) 'The following ERROR was detected: ', error
         STOP
      END IF
      WRITE(*,*) 'Number of nonzeros in factors = ',iparm(18)
      WRITE(*,*) 'Number of factorization MFLOPS = ',iparm(19)
C.. Factorization.
      phase = 22 ! only factorization
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
     1 idum, nrhs, iparm, msglvl, ddum, ddum, error)
      WRITE(*,*) 'Factorization completed ... '
      IF (error .NE. 0) THEN
         WRITE(*,*) 'The following ERROR was detected: ', error
         STOP
      ENDIF
C.. Back substitution and iterative refinement
      iparm(8) = 2 ! max numbers of iterative refinement steps
      phase = 33 ! only factorization
      do i = 1, n
         b(i) = 1.d0
      end do
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
     1 idum, nrhs, iparm, msglvl, b, x, error)
      WRITE(*,*) 'Solve completed ... '
      WRITE(*,*) 'The solution of the system is '
      DO i = 1, n
         WRITE(*,*) ' x(',i,') = ', x(i)
```

```
      END DO
C.. Termination and release of memory
      phase = -1 ! release internal memory
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, ddum, idum, idum,
     1 idum, nrhs, iparm, msglvl, ddum, ddum, error)
      END
```

Example C-9 Example C-9 Example pardiso_unsym.c for unsymmetric linear systems

```
/*
********************************************************************************
*
*                               INTEL CONFIDENTIAL
*   Copyright(C) 2004 Intel Corporation. All Rights Reserved.
*   The source code contained  or  described herein and all documents related to
*   the source code ("Material") are owned by Intel Corporation or its suppliers
*   or licensors.  Title to the  Material remains with  Intel Corporation or its
*   suppliers and licensors. The Material contains trade secrets and proprietary
*   and  confidential  information of  Intel or its suppliers and licensors. The
*   Material  is  protected  by  worldwide  copyright  and trade secret laws and
*   treaty  provisions. No part of the Material may be used, copied, reproduced,
*   modified, published, uploaded, posted, transmitted, distributed or disclosed
*   in any way without Intel's prior express written permission.
*   No license  under any  patent, copyright, trade secret or other intellectual
*   property right is granted to or conferred upon you by disclosure or delivery
*   of the Materials,  either expressly, by implication, inducement, estoppel or
*   otherwise.  Any  license  under  such  intellectual property  rights must be
*   express and approved by Intel in writing.
*
********************************************************************************
*
*   Content : MKL DSS C example
*
```

```
*******************************************************************************
*
*/
/* -------------------------------------------------------------------- */
/* Example program to show the use of the "PARDISO" routine */
/* on symmetric linear systems */
/* -------------------------------------------------------------------- */
/* This program can be downloaded from the following site: */
/* http://www.computational.unibas.ch/cs/scicomp */
/* */
/* (C) Olaf Schenk, Department of Computer Science, */
/* University of Basel, Switzerland. */
/* Email: olaf.schenk@unibas.ch */
/* -------------------------------------------------------------------- */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
extern int omp_get_max_threads();
/* PARDISO prototype. */
#if defined(_WIN32) || defined(_WIN64)
#define pardiso_ PARDISO
#else
#define PARDISO pardiso_
#endif
extern int PARDISO
        (void *, int *, int *, int *, int *, int *,
        double *, int *, int *, int *, int *, int *,
        int *, double *, double *, int *);

int main( void ) {
        /* Matrix data. */
        int n = 5;
        int ia[ 6] = { 1, 4, 6, 9, 12, 14 };
```

```
        int ja[13] = { 1, 2, 4,
                1, 2,
                3, 4, 5,
                1, 3, 4,
                2, 5 };
        double a[18] = { 1.0, -1.0, -3.0,
                -2.0, 5.0,
                4.0, 6.0, 4.0,
                -4.0, 2.0, 7.0,
                8.0, -5.0 };
        int mtype = 11; /* Real unsymmetric matrix */
        /* RHS and solution vectors. */
        double b[5], x[5];
        int nrhs = 1; /* Number of right hand sides. */
        /* Internal solver memory pointer pt, */
        /* 32-bit: int pt[64]; 64-bit: long int pt[64] */
        /* or void *pt[64] should be OK on both architectures */
        void *pt[64];
        /* Pardiso control parameters. */
        int iparm[64];
        int maxfct, mnum, phase, error, msglvl;
        /* Auxiliary variables. */
        int i;
        double ddum; /* Double dummy */
        int idum; /* Integer dummy. */
/* -------------------------------------------------------------------- */
/* .. Setup Pardiso control parameters. */
/* -------------------------------------------------------------------- */
        for (i = 0; i < 64; i++) {
                iparm[i] = 0;
        }
        iparm[0] = 1; /* No solver default */
        iparm[1] = 2; /* Fill-in reordering from METIS */
```

```
        /* Numbers of processors, value of OMP_NUM_THREADS */
        iparm[2] = omp_get_max_threads();
        iparm[3] = 0; /* No iterative-direct algorithm */
        iparm[4] = 0; /* No user fill-in reducing permutation */
        iparm[5] = 0; /* Write solution into x */
        iparm[6] = 0; /* Not in use */
        iparm[7] = 2; /* Max numbers of iterative refinement steps */
        iparm[8] = 0; /* Not in use */
        iparm[9] = 13; /* Perturb the pivot elements with 1E-13 */
        iparm[10] = 1; /* Use nonsymmetric permutation and scaling MPS */
        iparm[11] = 0; /* Not in use */
        iparm[12] = 0; /* Not in use */
        iparm[13] = 0; /* Output: Number of perturbed pivots */
        iparm[14] = 0; /* Not in use */
        iparm[15] = 0; /* Not in use */
        iparm[16] = 0; /* Not in use */
        iparm[17] = -1; /* Output: Number of nonzeros in the factor LU */
        iparm[18] = -1; /* Output: Mflops for LU factorization */
        iparm[19] = 0; /* Output: Numbers of CG Iterations */
        maxfct = 1; /* Maximum number of numerical factorizations. */
        mnum = 1; /* Which factorization to use. */
        msglvl = 1; /* Print statistical information in file */
        error = 0; /* Initialize error flag */
/* -------------------------------------------------------------------- */
/* .. Initialize the internal solver memory pointer. This is only */
/* necessary for the FIRST call of the PARDISO solver. */
/* -------------------------------------------------------------------- */
        for (i = 0; i < 64; i++) {
                pt[i] = 0;
        }
/* -------------------------------------------------------------------- */
/* .. Reordering and Symbolic Factorization. This step also allocates */
/* all memory that is necessary for the factorization. */
```

```
/* -------------------------------------------------------------------- */
        phase = 11;
        PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
                &n, a, ia, ja, &idum, &nrhs,
                iparm, &msglvl, &ddum, &ddum, &error);
        if (error != 0) {
                printf("\nERROR during symbolic factorization: %d", error);
                exit(1);
        }
        printf("\nReordering completed ... ");
        printf("\nNumber of nonzeros in factors = %d", iparm[17]);
        printf("\nNumber of factorization MFLOPS = %d", iparm[18]);
/* -------------------------------------------------------------------- */
/* .. Numerical factorization. */
/* -------------------------------------------------------------------- */
        phase = 22;
        PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
                &n, a, ia, ja, &idum, &nrhs,
                iparm, &msglvl, &ddum, &ddum, &error);
        if (error != 0) {
                printf("\nERROR during numerical factorization: %d", error);
                exit(2);
        }
        printf("\nFactorization completed ... ");
/* -------------------------------------------------------------------- */
/* .. Back substitution and iterative refinement. */
/* -------------------------------------------------------------------- */
        phase = 33;
        iparm[7] = 2; /* Max numbers of iterative refinement steps. */
        /* Set right hand side to one. */
        for (i = 0; i < n; i++) {
                b[i] = 1;
        }
```

```
        PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
                &n, a, ia, ja, &idum, &nrhs,
                iparm, &msglvl, b, x, &error);
        if (error != 0) {
                printf("\nERROR during solution: %d", error);
                exit(3);
        }
        printf("\nSolve completed ... ");
        printf("\nThe solution of the system is: ");
        for (i = 0; i < n; i++) {
                printf("\n x [%d] = % f", i, x[i] );
        }
        printf ("\n");
/* -------------------------------------------------------------------- */
/* .. Termination and release of memory. */
/* -------------------------------------------------------------------- */
        phase = -1; /* Release internal memory. */
        PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
                &n, &ddum, ia, ja, &idum, &nrhs,
                iparm, &msglvl, &ddum, &ddum, &error);
        return 0;
}
```

Direct Sparse Solver Examples

This section contains example code in Fortan 77, Fortran 90 and C. For description of the sparse solver routines used in this code, refer to "Direct Sparse Solver (DSS) Interface Routines" in Chapter 8 of the manual.

The example code solves the equations presented in Direct Method section of Appendix A - a symmetric positive definite system of equations $Ax = b$ with a sparse matrix, where

$$A = \begin{bmatrix} 9 & 1.5 & 6 & 0.75 & 3 \\ 1.5 & 0.5 & 0 & 0 & 0 \\ 6 & 0 & 12 & 0 & 0 \\ 0.75 & 0 & 0 & 0.625 & 0 \\ 3 & 0 & 0 & 0 & 16 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

Example results for symmetric systems

Upon successful execution of the solver, the determinant and the result of the solution array are as follows

```
pow of determinant is      0.000
base of determinant is      2.250
Determinant is      2.250
Solution Array:   -326.333   983.000   163.417   398.000    61.500
```

Example C-10 Fortran 77 example to solve symmetric positive definite system

```
*******************************************************************************
*
*                               INTEL CONFIDENTIAL
*   Copyright(C) 2001-2004 Intel Corporation. All Rights Reserved.
*   The source code contained  or  described herein and all documents related to
*   the source code ("Material") are owned by Intel Corporation or its suppliers
*   or licensors.  Title to the  Material remains with  Intel Corporation or its
*   suppliers and licensors. The Material contains trade secrets and proprietary
*   and  confidential  information of  Intel or its suppliers and licensors. The
*   Material  is  protected  by  worldwide  copyright  and trade secret laws and
*   treaty  provisions. No part of the Material may be used, copied, reproduced,
*   modified, published, uploaded, posted, transmitted, distributed or disclosed
*   in any way without Intel's prior express written permission.
*   No license  under any  patent, copyright, trade secret or other intellectual
*   property right is granted to or conferred upon you by disclosure or delivery
*   of the Materials,  either expressly, by implication, inducement, estoppel or
*   otherwise.  Any  license  under  such  intellectual property  rights must be
*   express and approved by Intel in writing.
*
*******************************************************************************
*
*   Content : Intel MKL DSS Fortran-77 example
```

```
*
*******************************************************************************
*
C----------------------------------------------------------------------------
C Example program for solving symmetric positive definite system of
C equations.
C----------------------------------------------------------------------------
      PROGRAM solver_f77_test
      IMPLICIT NONE
      INCLUDE 'mkl_dss.f77'
C----------------------------------------------------------------------------
C Define the array and rhs vectors
C----------------------------------------------------------------------------
      INTEGER nRows, nCols, nNonZeros, i, nRhs
      PARAMETER (nRows = 5,
     1 nCols = 5,
     2 nNonZeros = 9,
     3 nRhs = 1)
      INTEGER rowIndex(nRows + 1), columns(nNonZeros)
      DOUBLE PRECISION values(nNonZeros), rhs(nRows)
      DATA rowIndex / 1, 6, 7, 8, 9, 10 /
      DATA columns / 1, 2, 3, 4, 5, 2, 3, 4, 5 /
      DATA values / 9, 1.5, 6, .75, 3, 0.5, 12, .625, 16 /
      DATA rhs / 1, 2, 3, 4, 5 /
C----------------------------------------------------------------------------
C Allocate storage for the solver handle and the solution vector
C----------------------------------------------------------------------------
      DOUBLE PRECISION solution(nRows)
      INTEGER*8 handle
      INTEGER error
      CHARACTER*15 statIn
      DOUBLE PRECISION statOut(5)
      INTEGER bufLen
```

```
      PARAMETER(bufLen = 20)
      INTEGER buff(bufLen)
C---------------------------------------------------------------------------
C Initialize the solver
C---------------------------------------------------------------------------
      error = dss_create(handle, MKL_DSS_DEFAULTS)
      IF (error .NE. MKL_DSS_SUCCESS ) GOTO 999
C---------------------------------------------------------------------------
C Define the non-zero structure of the matrix
C---------------------------------------------------------------------------
      error = dss_define_structure( handle, MKL_DSS_SYMMETRIC,
     & rowIndex, nRows, nCols, columns, nNonZeros )
      IF (error .NE. MKL_DSS_SUCCESS ) GOTO 999
C---------------------------------------------------------------------------
C Reorder the matrix
C---------------------------------------------------------------------------
      error = dss_reorder( handle, MKL_DSS_DEFAULTS, 0)
      IF (error .NE. MKL_DSS_SUCCESS ) GOTO 999
C---------------------------------------------------------------------------
C Factor the matrix
C---------------------------------------------------------------------------
      error = dss_factor_real( handle,
     & MKL_DSS_DEFAULTS, VALUES)
      IF (error .NE. MKL_DSS_SUCCESS ) GOTO 999
C---------------------------------------------------------------------------
C Get the solution vector
C---------------------------------------------------------------------------
      error = dss_solve_real( handle, MKL_DSS_DEFAULTS,
     & rhs, nRhs, solution)
      IF (error .NE. MKL_DSS_SUCCESS ) GOTO 999
C---------------------------------------------------------------------------
C Print Determinant of the matrix
C---------------------------------------------------------------------------
```

```
      statIn = 'determinant'
      call mkl_cvt_to_null_terminated_str(buff,bufLen,statIn)
      error = dss_statistics(handle, MKL_DSS_DEFAULTS,
     & buff,statOut)
      WRITE(*,"(' pow of determinant is ', 5(F10.3))") statOut(1)
      WRITE(*,"(' base of determinant is ', 5(F10.3))") statOut(2)
      WRITE(*,"(' Determinant is ', 5(F10.3))")(10**statOut(1))*
     & statOut(2)
C---------------------------------------------------------------------------
C Deallocate solver storage
C---------------------------------------------------------------------------
      error = dss_delete( handle, MKL_DSS_DEFAULTS )
      IF (error .NE. MKL_DSS_SUCCESS ) GOTO 999
C---------------------------------------------------------------------------
C Print solution vector
C---------------------------------------------------------------------------
      WRITE(*,900) (solution(i), i = 1, nCols)
  900 FORMAT(' Solution Array: ',5(F10.3))
      GOTO 1000
  999 WRITE(*,*) "Solver returned error code ", error
 1000 END
```

Example C-11 C example to solve symmetric positive definite system

```
/*
*******************************************************************************
*
*                              INTEL CONFIDENTIAL
*   Copyright(C) 2001-2004 Intel Corporation. All Rights Reserved.
*   The source code contained  or  described herein and all documents related to
*   the source code ("Material") are owned by Intel Corporation or its suppliers
*   or licensors.  Title to the  Material remains with  Intel Corporation or its
*   suppliers and licensors. The Material contains trade secrets and proprietary
```

```
*   and  confidential  information of  Intel or its suppliers and licensors. The
*   Material  is  protected  by  worldwide  copyright  and trade secret laws and
*   treaty  provisions. No part of the Material may be used, copied, reproduced,
*   modified, published, uploaded, posted, transmitted, distributed or disclosed
*    in any way without Intel's prior express written permission.
*   No license  under any  patent, copyright, trade secret or other intellectual
*   property right is granted to or conferred upon you by disclosure or delivery
*   of the Materials,  either expressly, by implication, inducement, estoppel or
*   otherwise.  Any  license  under  such  intellectual property  rights must be
*    express and approved by Intel in writing.
*
********************************************************************************
*
*    Content : Intel MKL DSS C example
*
********************************************************************************
*/
/*
**
** Example program to solve symmetric positive definite system of equations.
**
*/
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include "mkl_dss.h"
/*
** Define the array and rhs vectors
*/
#define NROWS 5
#define NCOLS 5
#define NNONZEROS 9
#define NRHS 1
static const int nRows =     NROWS ;
```

```
static const int nCols =     NCOLS ;
static const int nNonZeros = NNONZEROS  ;
static const int nRhs =      NRHS ;
static _INTEGER_t rowIndex[NROWS+1] = { 1, 6, 7, 8, 9, 10 };
static _INTEGER_t columns[NNONZEROS] = { 1, 2, 3, 4, 5, 2, 3, 4, 5 };
static _DOUBLE_PRECISION_t values[NNONZEROS] = { 9, 1.5, 6, .75, 3, 0.5, 12, .625, 16 };
static _DOUBLE_PRECISION_t rhs[NCOLS] = { 1, 2, 3, 4, 5 };

void main() {
      int i;
      /* Allocate storage for the solver handle and the right-hand side. */
      _DOUBLE_PRECISION_t solValues[NROWS];
      _MKL_DSS_HANDLE_t handle;
      _INTEGER_t error;
      _CHARACTER_STR_t statIn[] = "determinant";
      _DOUBLE_PRECISION_t statOut[5];
      int opt = MKL_DSS_DEFAULTS;
      int sym = MKL_DSS_SYMMETRIC;
      int type = MKL_DSS_POSITIVE_DEFINITE;
/* --------------------- */
/* Initialize the solver */
/* --------------------- */
      error = dss_create(handle, opt );
      if ( error != MKL_DSS_SUCCESS ) goto printError;
/* ------------------------------------------- */
/* Define the non-zero structure of the matrix */
/* ------------------------------------------- */
      error = dss_define_structure(
            handle, sym, rowIndex, nRows, nCols,
            columns, nNonZeros );
      if ( error != MKL_DSS_SUCCESS ) goto printError;
/* ------------------ */
/* Reorder the matrix */
```

```
/* ------------------ */
      error = dss_reorder( handle, opt, 0);
      if ( error != MKL_DSS_SUCCESS ) goto printError;
/* ------------------ */
/* Factor the matrix */
/* ------------------ */
      error = dss_factor_real( handle, type, values );
      if ( error != MKL_DSS_SUCCESS ) goto printError;
/* ------------------------ */
/* Get the solution vector */
/* ------------------------ */
      error = dss_solve_real( handle, opt, rhs, nRhs, solValues );
      if ( error != MKL_DSS_SUCCESS ) goto printError;
/* ------------------------ */
/* Get the determinant */
/*--------------------------*/
      error = dss_statistics(handle, opt, statIn, statOut);
      if ( error != MKL_DSS_SUCCESS ) goto printError;
/*-------------------------*/
/* print determinant */
/*-------------------------*/
      printf(" determinant power is %g \n", statOut[0]);
      printf(" determinant base is %g \n", statOut[1]);
      printf(" Determinant is %g \n", (pow(10.0,statOut[0]))*statOut[1]);
      free((void *) statIn);
/* -------------------------- */
/* Deallocate solver storage */
/* -------------------------- */
      error = dss_delete( handle, opt );
      if ( error != MKL_DSS_SUCCESS ) goto printError;
/* ---------------------- */
/* Print solution vector */
/* ---------------------- */
```

```
	printf(" Solution array: ");
	for(i = 0; i< nCols; i++)
		printf(" %g", solValues[i] );
	printf("\n");
	exit(0);
printError:
	printf("Solver returned error code %d\n", error);
	exit(1);
}
```

Example C-12 Fortran 90 example to solve symmetric positive definite system

```
!*******************************************************************************
*
!                              INTEL CONFIDENTIAL
!   Copyright(C) 2001-2004 Intel Corporation. All Rights Reserved.
!   The source code contained  or  described herein and all documents related to
!   the source code ("Material") are owned by Intel Corporation or its suppliers
!   or licensors.  Title to the  Material remains with  Intel Corporation or its
!   suppliers and licensors. The Material contains trade secrets and proprietary
!   and  confidential  information of  Intel or its suppliers and licensors. The
!   Material  is  protected  by  worldwide  copyright  and trade secret laws and
!   treaty  provisions. No part of the Material may be used, copied, reproduced,
!   modified, published, uploaded, posted, transmitted, distributed or disclosed
!   in any way without Intel's prior express written permission.
!   No license  under any  patent, copyright, trade secret or other intellectual
!   property right is granted to or conferred upon you by disclosure or delivery
!   of the Materials,  either expressly, by implication, inducement, estoppel or
!   otherwise.  Any  license  under  such  intellectual property  rights must be
!   express and approved by Intel in writing.
!
!*******************************************************************************
*
!   Content : Intel MKL DSS Fortran-90 example
```

```
!
!*****************************************************************************
*
!---------------------------------------------------------------------------
!
! Example program for solving a symmetric positive definite system of
! equations.
!
!---------------------------------------------------------------------------
INCLUDE 'mkl_dss.f90' ! Include the standard DSS "header file."
PROGRAM solver_f90_test
use mkl_dss
IMPLICIT NONE
INTEGER, PARAMETER :: dp = KIND(1.0D0)
INTEGER :: error
INTEGER :: i
INTEGER, PARAMETER :: bufLen = 20
! Define the data arrays and the solution and rhs vectors.
INTEGER, ALLOCATABLE :: columns( : )
INTEGER :: nCols
INTEGER :: nNonZeros
INTEGER :: nRhs
INTEGER :: nRows
REAL(KIND=DP), ALLOCATABLE :: rhs( : )
INTEGER, ALLOCATABLE :: rowIndex( : )
REAL(KIND=DP), ALLOCATABLE :: solution( : )
REAL(KIND=DP), ALLOCATABLE :: values( : )
TYPE(MKL_DSS_HANDLE) :: handle ! Allocate storage for the solver handle.
REAL(KIND=DP),ALLOCATABLE::statOUt( : )
CHARACTER*15 statIn
INTEGER perm(1)
INTEGER buff(bufLen)
EXTERNAL MKL_CVT_TO_NULL_TERMINATED_STR
```

```
! Set the problem to be solved.
nRows = 5
nCols = 5
nNonZeros = 9
nRhs = 1
perm(1) = 0
ALLOCATE( rowIndex( nRows + 1 ) )
rowIndex = (/ 1, 6, 7, 8, 9, 10 /)
ALLOCATE( columns( nNonZeros ) )
columns = (/ 1, 2, 3, 4, 5, 2, 3, 4, 5 /)
ALLOCATE( values( nNonZeros ) )
values = (/ 9.0_DP, 1.5_DP, 6.0_DP, 0.75_DP, 3.0_DP, 0.5_DP, 12.0_DP, &
& 0.625_DP, 16.0_DP /)
ALLOCATE( rhs( nRows ) )
rhs = (/ 1.0_DP, 2.0_DP, 3.0_DP, 4.0_DP, 5.0_DP /)
! Initialize the solver.
error = dss_create( handle, MKL_DSS_DEFAULTS )
IF (error /= MKL_DSS_SUCCESS) GOTO 999
! Define the non-zero structure of the matrix.
error = dss_define_structure( handle, MKL_DSS_SYMMETRIC, rowIndex, nRows, &
& nCols, columns, nNonZeros )
IF (error /= MKL_DSS_SUCCESS) GOTO 999
! Reorder the matrix.
error = dss_reorder( handle, MKL_DSS_DEFAULTS, perm )
IF (error /= MKL_DSS_SUCCESS) GOTO 999
! Factor the matrix.
error = dss_factor_real( handle, MKL_DSS_DEFAULTS, values )
IF (error /= MKL_DSS_SUCCESS) GOTO 999
! Allocate the solution vector and solve the problem.
ALLOCATE( solution( nRows ) )
error = dss_solve_real(handle, MKL_DSS_DEFAULTS, rhs, nRhs, solution )
IF (error /= MKL_DSS_SUCCESS) GOTO 999
! Print Out the determinant of the matrix
```

```
ALLOCATE(statOut( 5 ) )
statIn = 'determinant'
call mkl_cvt_to_null_terminated_str(buff,bufLen,statIn);
error = dss_statistics(handle, MKL_DSS_DEFAULTS, buff, statOut )
IF (error /= MKL_DSS_SUCCESS) GOTO 999
WRITE(*,"('pow of determinant is '(5F10.3))") ( statOut(1) )
WRITE(*,"('base of determinant is '(5F10.3))") ( statOut(2) )
WRITE(*,"('Determinant is '(5F10.3))") ( (10**statOut(1))*statOut(2) )
! Deallocate solver storage and various local arrays.
error = dss_delete( handle, MKL_DSS_DEFAULTS )
IF (error /= MKL_DSS_SUCCESS ) GOTO 999
IF ( ALLOCATED( rowIndex) ) DEALLOCATE( rowIndex )
IF ( ALLOCATED( columns ) ) DEALLOCATE( columns )
IF ( ALLOCATED( values ) ) DEALLOCATE( values )
IF ( ALLOCATED( rhs ) ) DEALLOCATE( rhs )
IF ( ALLOCATED( statOut ) ) DEALLOCATE( statOut )
! Print the solution vector, deallocate it and exit
WRITE(*,"('Solution Array: '(5F10.3))") ( solution(i), i = 1, nCols )
IF ( ALLOCATED( solution ) ) DEALLOCATE( solution )
GOTO 1000
! Print an error message and exit
999 WRITE(*,*) "Solver returned error code ", error
1000 CONTINUE
END PROGRAM solver_f90_test
```

DFT Code Examples

This section presents code examples of using the DFT interface functions described in "Discrete Fourier Transform Functions" chapter.

Here are the examples of two one-dimensional computations. These examples use the default settings for all of the configuration parameters, which are specified in "Configuration Settings".

Example C-13 One-dimensional DFT (Fortran-interface)

```
! Fortran example.
! 1D complex to complex, and real to conjugate even
Use MKL_DFTI
Complex :: X(32)
Real :: Y(34)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc1_Handle, My_Desc2_Handle
Integer :: Status
...put input data into X(1),...,X(32); Y(1),...,Y(32)

! Perform a complex to complex transform
Status = DftiCreateDescriptor( My_Desc1_Handle, DFTI_SINGLE,
          DFTI_COMPLEX, 1, 32 )
Status = DftiCommitDescriptor( My_Desc1_Handle )
Status = DftiComputeForward( My_Desc1_Handle, X )
Status = DftiFreeDescriptor(My_Desc1_Handle)
! result is given by {X(1),X(2),...,X(32)}

! Perform a real to complex conjugate even transform
Status = DftiCreateDescriptor(My_Desc2_Handle, DFTI_SINGLE,
          DFTI_REAL, 1, 32)
Status = DftiCommitDescriptor(My_Desc2_Handle)
Status = DftiComputeForward(My_Desc2_Handle, Y)
Status = DftiFreeDescriptor(My_Desc2_Handle)
! result is given by {Y(1)+iY(2), Y(3)+iY(4), ..., Y(33)+iY(34),
! Y(31)-iY(32), Y(29)-iY(30), ..., Y(3)-iY(4).
```

Example C-14 One-dimensional DFT (C-interface)

```
/* C example, float _Complex is defined in C9X */
#include "mkl_dfti.h"
float _Complex x[32];
float y[34];
DFTI_DESCRIPTOR *my_desc1_handle, *my_desc2_handle;
/* .... or alternatively
DFTI_DESCRIPTOR_HANDLE my_desc1_handle, my_desc2_handle; */

long status;
...put input data into x[0],...,x[31]; y[0],...,y[31]
status = DftiCreateDescriptor( &my_desc1_handle, DFTI_SINGLE,
          DFTI_COMPLEX, 1, 32);
status = DftiCommitDescriptor( my_desc1_handle );
status = DftiComputeForward( my_desc1_handle, x);
status = DftiFreeDescriptor(&my_desc1_handle);
/* result is x[0], ..., x[31] */
status = DftiCreateDescriptor( &my_desc2_handle, DFTI_SINGLE,
          DFTI_REAL, 1, 32);
status = DftiCommitDescriptor( my_desc2_handle);
status = DftiComputeForward( my_desc2_handle, y);
status = DftiFreeDescriptor(&my_desc2_handle);
/* y[0]+iy[1], ..., y[32]+iy[33], y[30]-iy[31], ..., y[2]-iy[3] */
```

The following is an example of two simple two-dimensional transforms. Notice that the data and result parameters in computation functions are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. Therefore two-dimensional array must be transformed to one-dimensional array by `EQUIVALENCE` statement or other facilities of Fortran.

Example C-15 Two-dimensional DFT (Fortran-interface)

```
! Fortran example.
! 2D complex to complex, and real to conjugate even
Use MKL_DFTI
Complex ::  X_2D(32,100)
Real :: Y_2D(34, 102)
Complex ::  X(3200)
Real :: Y(3468)
Equivalence (X_2D, X)
Equivalence (Y_2D, Y)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc1_Handle, My_Desc2_Handle
Integer :: Status, L(2)
...put input data into X_2D(j,k), Y_2D(j,k), 1<=j=32,1<=k<=100
...set L(1) = 32, L(2) = 100
...the transform is a 32-by-100

! Perform a complex to complex transform
Status = DftiCreateDescriptor( My_Desc1_Handle, DFTI_SINGLE,
          DFTI_COMPLEX, 2, L)
Status = DftiCommitDescriptor( My_Desc1_Handle)
Status = DftiComputeForward( My_Desc1_Handle, X)
Status = DftiFreeDescriptor(My_Desc1_Handle)
! result is given by X_2D(j,k), 1<=j<=32, 1<=k<=100

! Perform a real to complex conjugate even transform
Status = DftiCreateDescriptor( My_Desc2_Handle, DFTI_SINGLE,
          DFTI_REAL, 2, L)
Status = DftiCommitDescriptor( My_Desc2_Handle)
Status = DftiComputeForward( My_Desc2_Handle, Y)
Status = DftiFreeDescriptor(My_Desc2_Handle)
! result is given by the complex value z(j,k) 1<=j<=32; 1<=k<=100 where
! z(j,k) = Y_2D(2j-1,k) + iY_2D(2j,k) 1<=j<=17; 1<=k<=100
! z(j,k) = Y_2D(2(34-j)-1,k) - iY_2D(2(34-j),k) 18<=j<=32; 1<=k<=100
```

Example C-16 Two-dimensional DFT (C-interface)

```
/* C example */
#include "mkl_dfti.h"
float _Complex x[32][100];
float y[34][102];
DFTI_DESCRIPTOR_HANDLE my_desc1_handle, my_desc2_handle;
/* or alternatively
DFTI_DESCRIPTOR *my_desc1_handle, *my_desc2_handle; */
long status, l[2];
...put input data into x[j][k] 0<=j<=31, 0<=k<=99
...put input data into y[j][k] 0<=j<=31, 0<=k<=99
l[0] = 32; l[1] = 100;
status = DftiCreateDescriptor( &my_desc1_handle, DFTI_SINGLE,
          DFTI_COMPLEX, 2, l);
status = DftiCommitDescriptor( my_desc1_handle);
status = DftiComputeForward( my_desc1_handle, x);
status = DftiFreeDescriptor(&my_desc1_handle);
/* result is the complex value x[j][k], 0<=j<=31, 0<=k<=99 */
status = DftiCreateDescriptor( &my_desc2_handle, DFTI_SINGLE,
          DFTI_REAL, 2, l);
status = DftiCommitDescriptor( my_desc2_handle);
status = DftiComputeForward( my_desc2_handle, y);
status = DftiFreeDescriptor(&my_desc2_handle);
/* result is the complex value z(j,k) 0<=j<=31; 0<=k<=99
/* z(j,k) = y[2j][k] + iy[2j+1][k] 0<=j<=16; 0<=k<=99 */
/* z(j,k) = y[2(32-j)][k] - iy[2(32-j)+1][k] 17<=j<=31; 1<=k<=100 */
```

The following examples demonstrate how you can change the default configuration settings by using the `DftiSetValue` function.

For instance, to preserve the input data after the DFT computation, the configuration of the `DFTI_PLACEMENT` should be changed to "not in place" from the default choice of "in place."

The code below illustrates how this can be done:

Example C-17 Changing Default Settings (Fortran)

```
! Fortran example
! 1D complex to complex, not in place
Use MKL_DFTI
Complex :: X_in(32), X_out(32)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc_Handle
Integer :: Status
...put input data into X_in(j), 1<=j<=32
Status = DftiCreateDescriptor( My_Desc_Handle, DFTI_SINGLE,
          DFTI_COMPLEX, 1, 32)
Status = DftiSetValue( My_Desc_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE)
Status = DftiCommitDescriptor( My_Desc_Handle)
Status = DftiComputeForward( My_Desc_Handle, X_in, X_out)
Status = DftiFreeDescriptor (My_Desc_Handle)
! result is X_out(1),X_out(2),...,X_out(32)
```

Example C-18 Changing Default Settings (C)

```
/* C example */
#include "mkl_dfti.h"
float  _Complex x_in[32], x_out[32];
DFTI_DESCRIPTOR_HANDLE my_desc_handle;
/* or alternatively
DFTI_DESCRIPTOR *my_desc_handle; */
long status;
...put input data into x_in[j], 0 <= j < 32
status = DftiCreateDescriptor( &my_desc_handle, DFTI_SINGLE,
DFTI_COMPLEX, 1, 32);
status = DftiSetValue( my_desc_handle, DFTI_PLACEMENT,
DFTI_NOT_INPLACE);
status = DftiCommitDescriptor( my_desc_handle);
status = DftiComputeForward( my_desc_handle, x_in, x_out);
status = DftiFreeDescriptor(&my_desc_handle);
/* result is x_out[0], x_out[1], ..., x_out[31] */
```

The Example C-19 below illustrates the use of the status checking functions described in Chapter 11.

Example C-19 Using Status Checking Function

```
from C language:

DFTI_DESCRIPTOR_HANDLE desc;
long status, class_error, value;
char* error_message;
 . . . descriptor creation and other code
status = DftiGetValue( desc, DFTI_PRECISION, &value); //
//or any DFTI function

class_error = DftiErrorClass(status, DFTI_ERROR_CLASS);
if (! class_error) {
printf ("status is not a member of Predefined Error
Class\n");
} else {
error_message = DftiErrorMessage(status);
printf("error_message = %s \n", error_message);
}
. . .
from Fortran:

type(DFTI_DESCRIPTOR), POINTER :: desc
integer value, status
character(DFTI_MAX_MESSAGE_LENGTH) error_message
logical class_error
 . . . descriptor creation and other code
status = DftiGetValue( desc, DFTI_PRECISION, value)

class_error = DftiErrorClass(status, DFTI_ERROR_CLASS)
if (.not. class_error) then
print *, 'status is not a member of Predefined Error
Class '
else
error_message  = DftiErrorMessage(status)
print *, 'error_message = ', error_message
endif
```

Below is an example where a 20-by-40 two-dimensional DFT is computed explicitly using one-dimensional transforms. Notice that the data and result parameters in computation functions are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. Therefore two-dimensional array must be transformed to one-dimensional array by `EQUIVALENCE` statement or other facilities of Fortran.

Example C-20 Computing 2D DFT by One-Dimentional Transforms

```
! Fortran
Complex ::  X_2D(20,40),
Complex ::  X(800)
Equivalence (X_2D, X)
INTEGER :: STRIDE(2)
type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle_Dim1
type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle_Dim2
...
Status = DftiCreateDescriptor( Desc_Handle_Dim1, DFTI_SINGLE,
                                 DFTI_COMPLEX, 1, 20 )
Status = DftiCreateDescriptor( Desc_Handle_Dim2, DFTI_SINGLE,
                                 DFTI_COMPLEX, 1, 40 )

! perform 40 one-dimensional transforms along 1st dimension
Status = DftiSetValue( Desc_Handle_Dim1, DFTI_NUMBER_OF_TRANSFORMS, 40 )
Status = DftiSetValue( Desc_Handle_Dim1, DFTI_INPUT_DISTANCE, 20 )
Status = DftiSetValue( Desc_Handle_Dim1, DFTI_OUTPUT_DISTANCE, 20 )
Status = DftiCommitDescriptor( Desc_Handle_Dim1 )
Status = DftiComputeForward( Desc_Handle_Dim1, X )

! perform 20 one-dimensional transforms along 2nd dimension
Stride(1) = 0; Stride(2) = 20
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_NUMBER_OF_TRANSFORMS, 20 )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_INPUT_DISTANCE, 1 )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_OUTPUT_DISTANCE, 1 )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_INPUT_STRIDES, Stride )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_OUTPUT_STRIDES, Stride )
```

```
Status = DftiCommitDescriptor( Desc_Handle_Dim2 )
Status = DftiComputeForward( Desc_Handle_Dim2, X )
Status = DftiFreeDescriptor( Desc_Handle_Dim1 )
Status = DftiFreeDescriptor( Desc_Handle_Dim2 )

/* C */
float _Complex x[20][40];
long stride[2];
DFTI_DESCRIPTOR_HANDLE Desc_Handle_Dim1;
DFTI_DESCRIPTOR_HANDLE Desc_Handle_Dim2;
...
status = DftiCreateDescriptor( &desc_handle_dim1, DFTI_SINGLE,
                                 DFTI_COMPLEX, 1, 20 );
status = DftiCreateDescriptor( &desc_handle_dim2, DFTI_SINGLE,
                                 DFTI_COMPLEX, 1, 40 );

/* perform 40 one-dimensional transforms along 1st dimension */
/* note that the 1st dimension data are not unit-stride */
stride[0] = 0; stride[1] = 40;
status = DftiSetValue( desc_handle_dim1, DFTI_NUMBER_OF_TRANSFORMS, 40 );
status = DftiSetValue( desc_handle_dim1, DFTI_INPUT_DISTANCE, 1 );
status = DftiSetValue( desc_handle_dim1, DFTI_OUTPUT_DISTANCE, 1 );
status = DftiSetValue( desc_handle_dim1, DFTI_INPUT_STRIDES, stride );
status = DftiSetValue( desc_handle_dim1, DFTI_OUTPUT_STRIDES, stride );
status = DftiCommitDescriptor( desc_handle_dim1 );
status = DftiComputeForward( desc_handle_dim1, x );

/* perform 20 one-dimensional transforms along 2nd dimension */
/* note that the 2nd dimension is unit stride */
status = DftiSetValue( desc_handle_dim2, DFTI_NUMBER_OF_TRANSFORMS, 20 );
status = DftiSetValue( desc_handle_dim2, DFTI_INPUT_DISTANCE, 40 );
status = DftiSetValue( desc_handle_dim2, DFTI_OUTPUT_DISTANCE, 40 );
status = DftiCommitDescriptor( desc_handle_dim2 );
status = DftiComputeForward( desc_handle_dim2, x );
```

```
status = DftiFreeDescriptor( &Desc_Handle_Dim1 );
status = DftiFreeDescriptor( &Desc_Handle_Dim2 );
```

Examples of Using Multi-Threading for DFT Computation

The following example program shows how to employ internal threading in Intel MKL for DFT computation (see case 1 in "Number of user threads").

To specify the number of threads inside Intel MKL, use the following settings:

`set OMP_NUM_THREADS = 1` for one-threaded mode;

`set OMP_NUM_THREADS = 4` for multi-threaded mode.

Note that the configuration parameter `DFTI_NUMBER_OF_USER_THREADS` must be equal to its default value 1 .

Example C-21 Using Intel MKL Internal Threading Mode

```
#include "mkl_dfti.h"

void main () {

float x[200][100];
DFTI_DESCRIPTOR_HANDLE my_desc1_handle;
long status, len[2];
//...put input data into x[j][k] 0<=j<=199, 0<=k<=99
len[0] = 200; len[1] = 100;
status = DftiCreateDescriptor( &my_desc1_handle, DFTI_SINGLE,DFTI_REAL, 2,
len);
status = DftiCommitDescriptor( my_desc1_handle);
status = DftiComputeForward( my_desc1_handle, x);
status = DftiFreeDescriptor(&my_desc1_handle);
}
```

The following Example C-22 illustrates a parallel customer program with each descriptor instance used only in a single thread (see case 2 in "Number of user threads").

To specify the number of threads, use the following settings:

`set MKL_SERIAL = 1` for single-threaded mode in Intel MKL (recommended);

`set OMP_NUM_THREADS = 4` for multi-threaded mode in customer program.

The configuration parameter `DFTI_NUMBER_OF_USER_THREADS` must be equal to its default value 1.

Note that in this example the program can be transformed to become single-threaded on the customer level but using parallel mode within Intel MKL. To achieve this, you need to set the parameter `DFTI_NUMBER_OF_TRANSFORMS = 4` and to set the corresponding parameter `DFTI_INPUT_DISTANCE = 5000`.

Example C-22 Using Parallel Mode with Multiple Descriptors

```
#include "mkl_dfti.h"
void main () {
float _Complex x[200][100];
DFTI_DESCRIPTOR_HANDLE my_desc_handle;
long status, len[2];
int iThread;
//...put input data into x[j][k] 0<=j<=199, 0<=k<=99
len[0] = 50; len[1] = 100;

int nThread = omp_get_max_threads();

// each thread calculates real DFT for matrix (50*100)
#pragma omp parallel default(shared)
{
#pragma omp for private(iThread, my_desc_handle)             /* parallel step */
for (iThread = 0; iThread < nThread; iThread++) {
    status = DftiCreateDescriptor( &my_desc_handle, DFTI_SINGLE, DFTI_COMPLEX, 2, len);
```

```
    status = DftiCommitDescriptor( my_desc_handle);
    status = DftiComputeForward( my_desc_handle, &x[iThread * len[0] * len[1]]);
    status = DftiFreeDescriptor(&my_desc_handle);
}/* parallel for */
}/* #pragma omp */
}
```

The following Example C-23 illustrates a parallel customer program with a common descriptor used in several threads (see case 3 in "Number of user threads").

In this case the number of threads, as well as any other configuration parameter, must not be changed after DFT initialization by the DftiCommitDescriptor() function is done.

Example C-23 Using Parallel Mode with a Common Descriptor

```
// set number of threads inside Intel MKL:
//rem set MKL_SERIAL = 1    -   is not required since one-threaded mode for
Intel MKL is forced automatically
// set OMP_NUM_THREADS = 4    -   multi-threaded mode for customer

#include "mkl_dfti.h"
void main () {
float _Complex x[200][100];
DFTI_DESCRIPTOR_HANDLE my_desc_handle;
long status, len[2];
int iThread;
//...put input data into x[j][k] 0<=j<=199, 0<=k<=99
len[0] = 50; len[1] = 100;

int nThread = omp_get_max_threads();

status = DftiCreateDescriptor( &my_desc_handle, DFTI_SINGLE, DFTI_COMPLEX, 2, len);
status = DftiSetValue(my_desc_handle, DFTI_NUMBER_OF_USER_THREADS, nThread);
```

```
status = DftiCommitDescriptor( my_desc_handle);

// each thread calculates real DFT for matrix (50*100)
#pragma omp parallel default(shared)
{
#pragma omp for private(iThread)             /* parallel step */
for (iThread = 0; iThread < nThread; iThread++) {
    status = DftiComputeForward( my_desc_handle, &x[iThread * len[0] * len[1]]);
}/* parallel for */
}
status = DftiFreeDescriptor(&my_desc_handle);
}
```

CBLAS Interface to the BLAS

D

This appendix presents CBLAS, the C interface to the Basic Linear Algebra Subprograms (BLAS) implemented in Intel® MKL.

Similar to BLAS, the CBLAS interface includes the following levels of functions:

- "Level 1 CBLAS" (vector-vector operations)
- "Level 2 CBLAS" (matrix-vector operations)
- "Level 3 CBLAS" (matrix-matrix operations).
- "Sparse CBLAS" (operations on sparse vectors).

To obtain the C interface, the Fortran routine names are prefixed with `cblas_` (for example, `dasum` becomes `cblas_dasum`). Names of all CBLAS functions are in lowercase letters.

Complex functions `?dotc` and `?dotu` become CBLAS subroutines (void functions); they return the complex result via a void pointer, added as the last parameter. CBLAS names of these functions are suffixed with _sub. For example, the BLAS function `cdotc` corresponds to `cblas_cdotc_sub`.

CBLAS Arguments

The arguments of CBLAS functions obey the following rules:

- Input arguments are declared with the `const` modifier.
- Non-complex scalar input arguments are passed by value.
- Complex scalar input arguments are passed as void pointers.
- Array arguments are passed by address.
- Output scalar arguments are passed by address.

- BLAS character arguments are replaced by the appropriate enumerated type.
- Level 2 and Level 3 routines acquire an additional parameter of type `CBLAS_ORDER` as their first argument. This parameter specifies whether two-dimensional arrays are row-major (`CblasRowMajor`) or column-major (`CblasColMajor`).

Enumerated Types

The CBLAS interface uses the following enumerated types:

```
enum CBLAS_ORDER {
   CblasRowMajor=101,  /* row-major arrays */
   CblasColMajor=102};   /* column-major arrays */
enum CBLAS_TRANSPOSE {
   CblasNoTrans=111,     /* trans='N' */
   CblasTrans=112,       /* trans='T' */
   CblasConjTrans=113};  /* trans='C' */
enum CBLAS_UPLO {
   CblasUpper=121,        /* uplo ='U' */
   CblasLower=122};       /* uplo ='L' */
enum CBLAS_DIAG {
   CblasNonUnit=131,      /* diag ='N' */
   CblasUnit=132};        /* diag ='U' */
enum CBLAS_SIDE {
   CblasLeft=141,         /* side ='L' */
   CblasRight=142};       /* side ='R' */
```

Level 1 CBLAS

This is an interface to "BLAS Level 1 Routines and Functions", which perform basic vector-vector operations.

ipps?asum

```
float cblas_sasum(const int N, const float *X, const int incX);
double cblas_dasum(const int N, const double *X, const int incX);
float cblas_scasum(const int N, const void *X, const int incX);
double cblas_dzasum(const int N, const void *X, const int incX);
```

ipps?axpy

```
void cblas_saxpy(const int N, const float alpha, const float *X, const int incX,
float *Y, const int incY);
void cblas_daxpy(const int N, const double alpha, const double *X, const int
incX, double *Y, const int incY);
void cblas_caxpy(const int N, const void *alpha, const void *X, const int incX,
void *Y, const int incY);
void cblas_zaxpy(const int N, const void *alpha, const void *X, const int incX,
void *Y, const int incY);
```

ipps?copy

```
void cblas_scopy(const int N, const float *X, const int incX, float *Y, const int
incY);
void cblas_dcopy(const int N, const double *X, const int incX, double *Y, const
int incY);
void cblas_ccopy(const int N, const void *X, const int incX, void *Y, const int
incY);
void cblas_zcopy(const int N, const void *X, const int incX, void *Y, const int
incY);
```

ipps?dot

```
float cblas_sdot(const int N, const float *X, const int incX,
const float *Y, const int incY);
double cblas_ddot(const int N, const double *X, const int incX,
const double *Y, const int incY);
```

ipps?sdot

```
float cblas_sdsdot(const int N, const float *SB, const float *SX, const int incX,
const float *SY, const int incY);
double cblas_dsdot(const int N, const float *SX, const int incX, const float *SY,
const int incY);
```

ipps?dotc

```
void cblas_cdotc_sub(const int N, const void *X, const int incX, const void *Y,
const int incY, void *dotc);
void cblas_zdotc_sub(const int N, const void *X, const int incX, const void *Y,
const int incY, void *dotc);
```

ipps?dotu

```
void cblas_cdotu_sub(const int N, const void *X, const int incX, const void *Y,
const int incY, void *dotu);
void cblas_zdotu_sub(const int N, const void *X, const int incX, const void *Y,
const int incY, void *dotu);
```

ipps?nrm2

```
float cblas_snrm2(const int N, const float *X, const int incX);
double cblas_dnrm2(const int N, const double *X, const int incX);
float cblas_scnrm2(const int N, const void *X, const int incX);
double cblas_dznrm2(const int N, const void *X, const int incX);
```

ipps?rot

```
void cblas_srot(const int N, float *X, const int incX, float *Y, const int incY,
const float c, const float s);
void cblas_drot(const int N, double *X, const int incX, double *Y,const int incY,
const double c, const double s);
```

ipps?rotg

```
void cblas_srotg(float *a, float *b, float *c, float *s);
void cblas_drotg(double *a, double *b, double *c, double *s);
```

ipps?rotm

```
void cblas_srotm(const int N, float *X, const int incX, float *Y, const int incY,
const float *P);
void cblas_drotm(const int N, double *X, const int incX, double *Y, const int
incY, const double *P);
```

ipps?rotmg

```
void cblas_srotmg(float *d1, float *d2, float *b1, const float b2, float *P);
void cblas_drotmg(double *d1, double *d2, double *b1, const double b2, double
*P);
```

ipps?scal

```
void cblas_sscal(const int N, const float alpha, float *X, const int incX);
void cblas_dscal(const int N, const double alpha, double *X, const int incX);
void cblas_cscal(const int N, const void *alpha, void *X, const int incX);
void cblas_zscal(const int N, const void *alpha, void *X, const int incX);
void cblas_csscal(const int N, const float alpha, void *X, const int incX);
void cblas_zdscal(const int N, const double alpha, void *X, const int incX);
```

ipps?swap

```
void cblas_sswap(const int N, float *X, const int incX, float *Y, const int incY);
void cblas_dswap(const int N, double *X, const int incX, double *Y, const int
incY);
void cblas_cswap(const int N, void *X, const int incX, void *Y, const int incY);
void cblas_zswap(const int N, void *X, const int incX, void *Y, const int incY);
```

ippsi?amax

```
CBLAS_INDEX cblas_isamax(const int N, const float *X, const int incX);
CBLAS_INDEX cblas_idamax(const int N, const double *X, const int incX);
CBLAS_INDEX cblas_icamax(const int N, const void *X, const int incX);
CBLAS_INDEX cblas_izamax(const int N, const void *X, const int incX);
```

ippsi?amin

```
CBLAS_INDEX cblas_isamin(const int N, const float *X, const int incX);
CBLAS_INDEX cblas_idamin(const int N, const double *X, const int incX);
CBLAS_INDEX cblas_icamin(const int N, const void *X, const int incX);
CBLAS_INDEX cblas_izamin(const int N, const void *X, const int incX);
```

Level 2 CBLAS

This is an interface to "BLAS Level 2 Routines", which perform basic matrix-vector operations. Each C routine in this group has an additional parameter of type `CBLAS_ORDER` (the first argument) that determines whether the two-dimensional arrays use column-major or row-major storage.

ipps?gbmv

```
void cblas_sgbmv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA,
const int M, const int N, const int KL, const int KU, const float alpha, const
float *A, const int lda, const float *X, const int incX, const float beta, float
*Y, const int incY);
```

```
void cblas_dgbmv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA,
const int M, const int N, const int KL, const int KU, const double alpha, const
double *A, const int lda, const double *X, const int incX, const double beta,
double *Y, const int incY);
```

```
void cblas_cgbmv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA,
const int M, const int N, const int KL, const int KU, const void *alpha, const
void *A, const int lda, const void *X, const int incX, const void *beta, void *Y,
const int incY);
```

```
void cblas_zgbmv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA,
const int M, const int N, const int KL, const int KU, const void *alpha, const
void *A, const int lda, const void *X, const int incX, const void *beta, void *Y,
const int incY);
```

ipps?gemv

```
void cblas_sgemv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA,
const int M, const int N, const float alpha, const float *A, const int lda, const
float *X, const int incX, const float beta, float *Y, const int incY);
```

```
void cblas_dgemv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA,
const int M, const int N, const double alpha, const double *A, const int lda,
const double *X, const int incX, const double beta, double *Y, const int incY);
```

```
void cblas_cgemv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA,
const int M, const int N, const void *alpha, const void *A, const int lda, const
void *X, const int incX, const void *beta, void *Y, const int incY);
```

```
void cblas_zgemv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE TransA,
const int M, const int N, const void *alpha, const void *A, const int lda, const
void *X, const int incX, const void *beta, void *Y, const int incY);
```

ipps?ger

```
void cblas_sger(const enum CBLAS_ORDER order, const int M, const int N, const
float alpha, const float *X, const int incX, const float *Y, const int incY, float
*A, const int lda);
```

```
void cblas_dger(const enum CBLAS_ORDER order, const int M, const int N, const
double alpha, const double *X, const int incX, const double *Y, const int incY,
double *A, const int lda);
```

ipps?gerc

```
void cblas_cgerc(const enum CBLAS_ORDER order, const int M, const int N, const
void *alpha, const void *X, const int incX, const void *Y, const int incY, void
*A, const int lda);
```

```
void cblas_zgerc(const enum CBLAS_ORDER order, const int M, const int N, const
void *alpha, const void *X, const int incX, const void *Y, const int incY, void
*A, const int lda);
```

ipps?geru

```
void cblas_cgeru(const enum CBLAS_ORDER order, const int M, const int N, const
void *alpha, const void *X, const int incX, const void *Y, const int incY, void
*A, const int lda);
```

```
void cblas_zgeru(const enum CBLAS_ORDER order, const int M, const int N, const
void *alpha, const void *X, const int incX, const void *Y, const int incY, void
*A, const int lda);
```

ipps?hbmv

```
void cblas_chbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const int K, const void *alpha, const void *A, const int lda, const void
*X, const int incX, const void *beta, void *Y, const int incY);
```

```
void cblas_zhbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const int K, const void *alpha, const void *A, const int lda, const void
*X, const int incX, const void *beta, void *Y, const int incY);
```

ipps?hemv

```
void cblas_chemv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const void *alpha, const void *A, const int lda, const void *X, const int
incX, const void *beta, void *Y, const int incY);
```

```
void cblas_zhemv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const void *alpha, const void *A, const int lda, const void *X, const int
incX, const void *beta, void *Y, const int incY);
```

ipps?her

```
void cblas_cher(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const float alpha, const void *X, const int incX, void *A, const int lda);
```

```
void cblas_zher(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const double alpha, const void *X, const int incX, void *A, const int lda);
```

ipps?her2

```
void cblas_cher2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const void *alpha, const void *X, const int incX, const void *Y, const int
incY, void *A, const int lda);
```

```
void cblas_zher2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const void *alpha, const void *X, const int incX, const void *Y, const int
incY, void *A, const int lda);
```

ipps?hpmv

```
void cblas_chpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const void *alpha, const void *Ap, const void *X, const int incX, const
void *beta, void *Y, const int incY);
```

```
void cblas_zhpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const void *alpha, const void *Ap, const void *X, const int incX, const
void *beta, void *Y, const int incY);
```

ipps?hpr

```
void cblas_chpr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const float alpha, const void *X, const int incX, void *A);
void cblas_zhpr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const double alpha, const void *X, const int incX, void *A);
```

ipps?hpr2

```
void cblas_chpr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const void *alpha, const void *X, const int incX, const void *Y, const int
incY, void *Ap);
void cblas_zhpr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const void *alpha, const void *X, const int incX, const void *Y, const int
incY, void *Ap);
```

ipps?sbmv

```
void cblas_ssbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const int K, const float alpha, const float *A, const int lda, const float
*X, const int incX, const float beta, float *Y, const int incY);
void cblas_dsbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const int K, const double alpha, const double *A, const int lda, const
double *X, const int incX, const double beta, double *Y, const int incY);
```

ipps?spmv

```
void cblas_sspmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const float alpha, const float *Ap, const float *X, const int incX, const
float beta, float *Y, const int incY);
void cblas_dspmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const double alpha, const double *Ap, const double *X, const int incX,
const double beta, double *Y, const int incY);
```

ipps?spr

```
void cblas_sspr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const float alpha, const float *X, const int incX, float *Ap);
void cblas_dspr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const double alpha, const double *X, const int incX, double *Ap);
```

ipps?spr2

```
void cblas_sspr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const float alpha, const float *X, const int incX, const float *Y, const
int incY, float *A);
```

```
void cblas_dspr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const double alpha, const double *X, const int incX, const double *Y, const
int incY, double *A);
```

ipps?symv

```
void cblas_ssymv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const float alpha, const float *A, const int lda, const float *X, const int
incX, const float beta, float *Y, const int incY);
```

```
void cblas_dsymv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const double alpha, const double *A, const int lda, const double *X, const
int incX, const double beta, double *Y, const int incY);
```

ipps?syr

```
void cblas_ssyr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const float alpha, const float *X, const int incX, float *A, const int
lda);
```

```
void cblas_dsyr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const double alpha, const double *X, const int incX, double *A, const int
lda);
```

ipps?syr2

```
void cblas_ssyr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const float alpha, const float *X, const int incX, const float *Y, const
int incY, float *A, const int lda);
```

```
void cblas_dsyr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
int N, const double alpha, const double *X, const int incX, const double *Y, const
int incY, double *A, const int lda);
```

ipps?tbmv

```
void cblas_stbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int
K, const float *A, const int lda, float *X, const int incX);
```

```
void cblas_dtbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int
K, const double *A, const int lda, double *X, const int incX);
```

```
void cblas_ctbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int
K, const void *A, const int lda, void *X, const int incX);
```

```
void cblas_ztbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int
K, const void *A, const int lda, void *X, const int incX);
```

ipps?tbsv

```
void cblas_stbsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int
K, const float *A, const int lda, float *X, const int incX);
```

```
void cblas_dtbsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int
K, const double *A, const int lda, double *X, const int incX);
```

```
void cblas_ctbsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int
K, const void *A, const int lda, void *X, const int incX);
```

```
void cblas_ztbsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const int
K, const void *A, const int lda, void *X, const int incX);
```

ipps?tpmv

```
void cblas_stpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const float
*Ap, float *X, const int incX);
```

```
void cblas_dtpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N, const
double *Ap, double *X, const int incX);
```

```
void cblas_ctpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,const void
*Ap, void *X, const int incX);
```

```
void cblas_ztpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,const void
*Ap, void *X, const int incX);
```

ipps?tpsv

```
void cblas_stpsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,const float
*Ap, float *X, const int incX);
```

```
void cblas_dtpsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,const double
*Ap, double *X, const int incX);
```

```
void cblas_ctpsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,const void
*Ap, void *X, const int incX);
```

```
void cblas_ztpsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,const void
*Ap, void *X, const int incX);
```

ipps?trmv

```
void cblas_strmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,const float
*A, const int lda, float *X, const int incX);
```

```
void cblas_dtrmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,const double
*A, const int lda, double *X, const int incX);
```

```
void cblas_ctrmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,const void
*A, const int lda, void *X, const int incX);
```

```
void cblas_ztrmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,const void
*A, const int lda, void *X, const int incX);
```

ipps?trsv

```
void cblas_strsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,const float
*A, const int lda, float *X, const int incX);
```

```
void cblas_dtrsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,const double
*A, const int lda, double *X, const int incX);
```

```
void cblas_ctrsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,const void
*A, const int lda, void *X, const int incX);
```

```
void cblas_ztrsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,const void
*A, const int lda, void *X, const int incX);
```

Level 3 CBLAS

This is an interface to "BLAS Level 3 Routines", which perform basic matrix-matrix operations. Each C routine in this group has an additional parameter of type `CBLAS_ORDER` (the first argument) that determines whether the two-dimensional arrays use column-major or row-major storage.

ipps?gemm

```
void cblas_sgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA,
const enum CBLAS_TRANSPOSE TransB, const int M, const int N, const int K, const
float alpha, const float *A, const int lda, const float *B, const int ldb, const
float beta, float *C, const int ldc);
```

```
void cblas_dgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA,
const enum CBLAS_TRANSPOSE TransB, const int M, const int N, const int K, const
double alpha, const double *A, const int lda, const double *B, const int ldb,
const double beta, double *C, const int ldc);
```

```
void cblas_cgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA,
const enum CBLAS_TRANSPOSE TransB, const int M, const int N, const int K, const
void *alpha, const void *A, const int lda, const void *B, const int ldb, const
void *beta, void *C, const int ldc);
```

```
void cblas_zgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA,
const enum CBLAS_TRANSPOSE TransB, const int M, const int N, const int K, const
void *alpha, const void *A, const int lda, const void *B, const int ldb, const
void *beta, void *C, const int ldc);
```

ipps?hemm

```
void cblas_chemm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const
enum CBLAS_UPLO Uplo, const int M, const int N, const void *alpha, const void *A,
const int lda, const void *B, const int ldb, const void *beta, void *C, const int
ldc);
```

```
void cblas_zhemm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const
enum CBLAS_UPLO Uplo, const int M, const int N, const void *alpha, const void *A,
const int lda, const void *B, const int ldb, const void *beta, void *C, const int
ldc);
```

ipps?herk

```
void cblas_cherk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE Trans, const int N, const int K, const float alpha, const
void *A, const int lda, const float beta, void *C, const int ldc);
```

```
void cblas_zherk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE Trans, const int N, const int K, const double alpha, const
void *A, const int lda, const double beta, void *C, const int ldc);
```

ipps?her2k

```
void cblas_cher2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE Trans, const int N, const int K, const void *alpha, const
void *A, const int lda, const void *B, const int ldb, const float beta, void *C,
const int ldc);
```

```
void cblas_zher2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE Trans, const int N, const int K, const void *alpha, const
void *A, const int lda, const void *B, const int ldb, const double beta, void *C,
const int ldc);
```

ipps?symm

```
void cblas_ssymm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const
enum CBLAS_UPLO Uplo, const int M, const int N, const float alpha, const float *A,
const int lda, const float *B, const int ldb, const float beta, float *C, const
int ldc);
```

```
void cblas_dsymm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const
enum CBLAS_UPLO Uplo, const int M, const int N, const double alpha, const double
*A, const int lda, const double *B, const int ldb, const double beta, double *C,
const int ldc);
```

```
void cblas_csymm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const
enum CBLAS_UPLO Uplo, const int M, const int N, const void *alpha, const void *A,
const int lda, const void *B, const int ldb, const void *beta, void *C, const int
ldc);
```

```
void cblas_zsymm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const
enum CBLAS_UPLO Uplo, const int M, const int N, const void *alpha, const void *A,
const int lda, const void *B, const int ldb, const void *beta, void *C, const int
ldc);
```

ipps?syrk

```
void cblas_ssyrk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE Trans, const int N, const int K, const float alpha, const
float *A, const int lda, const float beta, float *C, const int ldc);
```

```
void cblas_dsyrk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE Trans, const int N, const int K, const double alpha, const
double *A, const int lda, const double beta, double *C, const int ldc);
```

```
void cblas_csyrk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE Trans, const int N, const int K, const void *alpha, const
void *A, const int lda, const void *beta, void *C, const int ldc);
```

```
void cblas_zsyrk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE Trans, const int N, const int K, const void *alpha, const
void *A, const int lda, const void *beta, void *C, const int ldc);
```

ipps?syr2k

```
void cblas_ssyr2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE Trans, const int N, const int K, const float alpha,const
float *A, const int lda, const float *B, const int ldb, const float beta, float
*C, const int ldc);
```

```
void cblas_dsyr2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE Trans, const int N, const int K, const double alpha, const
double *A, const int lda, const double *B, const int ldb, const double beta,
double *C, const int ldc);
```

```
void cblas_csyr2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSP SE Trans, const int N, const int K, const void *alpha,const void
*A, const int lda, const void *B, const int ldb, const void *beta, void *C, const
int ldc);
```

```
void cblas_zsyr2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo, const
enum CBLAS_TRANSPOSE Trans, const int N, const int K, const void *alpha, const
void *A, const int lda, const void *B, const int ldb, const void *beta, void *C,
const int ldc);
```

ipps?trmm

```
void cblas_strmm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const
enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG
Diag, const int M, const int N, const float alpha, const float *A, const int lda,
float *B, const int ldb);
```

```
void cblas_dtrmm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const
enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG
Diag, const int M, const int N, const double alpha, const double *A, const int
lda, double *B, const int ldb);
```

```
void cblas_ctrmm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const
enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG
Diag, const int M, const int N, const void *alpha, const void *A, const int lda,
void *B, const int ldb);
```

```
void cblas_ztrmm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const
enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG
Diag, const int M, const int N, const void *alpha, const void *A, const int lda,
void *B, const int ldb);
```

ipps?trsm

```
void cblas_strsm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const
enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG
Diag, const int M, const int N, const float alpha, const float *A, const int lda,
float *B, const int ldb);
```

```
void cblas_dtrsm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const
enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG
Diag, const int M, const int N, const double alpha, const double *A, const int
lda, double *B, const int ldb);
```

```
void cblas_ctrsm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const
enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG
Diag, const int M, const int N, const void *alpha, const void *A, const int lda,
void *B, const int ldb);
void cblas_ztrsm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side, const
enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG
Diag, const int M, const int N, const void *alpha, const void *A, const int lda,
void *B, const int ldb);
```

Sparse CBLAS

This is an interface to "Sparse BLAS Routines and Functions", which perform a number of common vector operations on sparse vectors stored in compressed form.

Note that all index parameters, *indx*, are in C-type notation and vary in the range [0..*N*-1].

ipps?axpyi

```
void cblas_saxpyi(const int N, const float alpha,
const float *X, const int *indx, float *Y);
void cblas_daxpyi(const int N, const double alpha,
const double *X, const int *indx, double *Y);
void cblas_caxpyi(const int N, const void *alpha,
const void *X, const int *indx, void *Y);
void cblas_zaxpyi(const int N, const void *alpha,
const void *X, const int *indx, void *Y);
```

ipps?doti

```
float  cblas_sdoti(const int N, const float *X,
const int *indx, const float *Y);
double cblas_ddoti(const int N, const double *X,
const int *indx, const double *Y);
```

ipps?dotci

```
void   cblas_cdotci_sub(const int N, const void *X, const int *indx, const void
*Y, void *dotui);
void   cblas_zdotci_sub(const int N, const void *X, const int *indx, const void
*Y, void *dotui);
```

ipps?dotui

```
void   cblas_cdotui_sub(const int N, const void *X, const int *indx, const void
*Y, void *dotui);
void   cblas_zdotui_sub(const int N, const void *X, const int *indx, const void
*Y, void *dotui);
```

ipps?gthr

```
void cblas_sgthr(const int N, const float *Y, float *X,
const int *indx);
void cblas_dgthr(const int N, const double *Y, double *X,
const int *indx);
void cblas_cgthr(const int N, const void *Y, void *X,
const int *indx);
```

```
void cblas_zgthr(const int N, const void *Y, void *X,
const int *indx);
```

ipps?gthrz

```
void cblas_sgthrz(const int N, float *Y, float *X,
const int *indx);
void cblas_dgthrz(const int N, double *Y, double *X,
const int *indx);
void cblas_cgthrz(const int N, void *Y, void *X,
const int *indx);
void cblas_zgthrz(const int N, void *Y, void *X,
const int *indx);
```

ipps?roti

```
void cblas_sroti(const int N, float *X, const int *indx,
float *Y, const float c, const float s);
void cblas_droti(const int N, double *X, const int *indx,
double *Y, const double c, const double s);
```

ipps?sctr

```
void cblas_ssctr(const int N, const float *X, const int *indx, float *Y);
void cblas_dsctr(const int N, const double *X, const int *indx, double *Y);
void cblas_csctr(const int N, const void *X, const int *indx, void *Y);
void cblas_zsctr(const int N, const void *X, const int *indx, void *Y);
```

Glossary

A^H	Denotes the conjugate of a general matrix A. *See also* conjugate matrix.
A^T	Denotes the transpose of a general matrix A. *See also* transpose.
band matrix	A general m by n matrix A such that $a_{ij} = 0$ for $\|i - j\| > l$, where $1 < l < \min(m, n)$. For example, any tridiagonal matrix is a band matrix.
band storage	A special storage scheme for band matrices. A matrix is stored in a two-dimensional array: columns of the matrix are stored in the corresponding columns of the array, and *diagonals* of the matrix are stored in rows of the array.
BLAS	Abbreviation for Basic Linear Algebra Subprograms. These subprograms implement vector, matrix-vector, and matrix-matrix operations.
BRNG	Abbreviation for Basic Random Number Generator. Basic random number generators are pseudorandom number generators imitating i.i.d. random number sequences of uniform distribution. Distributions other than uniform are generated by applying different transformation techniques to the sequences of random numbers of uniform distribution.
BRNG registration	Standardized mechanism that allows a user to include a user-designed BRNG into the VSL and use it along with the predefined VSL basic generators.

Bunch-Kaufman factorization	Representation of a real symmetric or complex Hermitian matrix A in the form $A = PUDU^HP^T$ (or $A = PLDL^HP^T$) where P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D.
c	When found as the first letter of routine names, `c` indicates the usage of single-precision complex data type.
CBLAS	C interface to the BLAS. *See* BLAS.
CDF	Cumulative Distribution Function. The function that determines probability distribution for univariate or multivariate random variable X. For univariate distribution the cumulative distribution function is the function of real argument x, which for every x takes a value equal to probability of the event A: $X \le x$. For multivariate distribution the cumulative distribution function is the function of a real vector $x = (x_1, x_2, ..., x_n)$, which, for every x, takes a value equal to probability of the event $A = (X_1 \le x_1$ & $X_2 \le x_2$, & ..., & $X_n \le x_n)$.
Cholesky factorization	Representation of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A in the form $A = U^HU$ or $A = LL^H$, where L is a lower triangular matrix and U is an upper triangular matrix.
condition number	The number $\kappa(A)$ defined for a given square matrix A as follows: $\kappa(A) = \|A\| \|A^{-1}\|$.
conjugate matrix	The matrix A^H defined for a given general matrix A as follows: $(A^H)_{ij} = (a_{ji})^*$.
conjugate number	The conjugate of a complex number $z = a + bi$ is $z^* = a - bi$.

d	When found as the first letter of routine names, `d` indicates the usage of double-precision real data type.
dot product	The number denoted $x \cdot y$ and defined for given vectors x and y as follows: $x \cdot y = \Sigma_i x_i y_i$. Here x_i and y_i stand for the ith elements of x and y, respectively.
double precision	A floating-point data type. On Intel® processors, this data type allows you to store real numbers x such that $2.23*10^{-308} < \| x \| < 1.79*10^{308}$. For this data type, the machine precision ε is approximately 10^{-15}, which means that double-precision numbers usually contain no more than 15 significant decimal digits. For more information, refer to *Pentium® Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual*.
eigenvalue	*See* eigenvalue problem.
eigenvalue problem	A problem of finding non-zero vectors x and numbers λ (for a given square matrix A) such that $Ax = \lambda x$. Here the numbers λ are called the *eigenvalues* of the matrix A and the vectors x are called the *eigenvectors* of the matrix A.
eigenvector	*See* eigenvalue problem.
elementary reflector (Householder matrix)	Matrix of a general form $H = I - \tau vv^T$, where v is a column vector and τ is a scalar. In LAPACK elementary reflectors are used, for example, to represent the matrix Q in the QR factorization (the matrix Q is represented as a product of elementary reflectors).
factorization	Representation of a matrix as a product of matrices. *See also* Bunch-Kaufman factorization, Cholesky factorization, *LU* factorization, *LQ* factorization, *QR* factorization, Schur factorization.

FFTs — Abbreviation for Fast Fourier Transforms. *See* Chapter 3 of this book.

full storage — A storage scheme allowing you to store matrices of any kind. A matrix A is stored in a two-dimensional array `a`, with the matrix element a_{ij} stored in the array element `a(i,j)`.

Hermitian matrix — A square matrix A that is equal to its conjugate matrix A^H. The conjugate A^H is defined as follows: $(A^H)_{ij} = (a_{ji})^*$.

I — *See* identity matrix.

identity matrix — A square matrix I whose diagonal elements are 1, and off-diagonal elements are 0. For any matrix A, $AI = A$ and $IA = A$.

i.i.d. — Independent Identically Distributed.

in-place — Qualifier of an operation. A function that performs its operation in-place takes its input from an array and returns its output to the same array.

Intel MKL — Abbreviation for Intel® Math Kernel Library.

inverse matrix — The matrix denoted as A^{-1} and defined for a given square matrix A as follows: $AA^{-1} = A^{-1}A = I$. A^{-1} does not exist for singular matrices A.

LQ factorization — Representation of an m by n matrix A as $A = LQ$ or $A = (L\ 0)Q$. Here Q is an n by n orthogonal (unitary) matrix. For $m \le n$, L is an m by m lower triangular matrix with real diagonal elements; for $m > n$,

$$L = \begin{bmatrix} L_1 \\ L_2 \end{bmatrix}$$ where L_1 is an n by n lower triangular matrix, and L_2 is a rectangular matrix.

LU factorization — Representation of a general m by n matrix A as $A = PLU$, where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$).

machine precision
: The number ε determining the precision of the machine representation of real numbers. For Intel® architecture, the machine precision is approximately 10^{-7} for single-precision data, and approximately 10^{-15} for double-precision data. The precision also determines the number of significant decimal digits in the machine representation of real numbers. *See also* double precision and single precision.

MPI
: Message Passing Interface. This standard defines the user interface and functionality for a wide range of message-passing capabilities in parallel computing.

MPICH
: A freely available, portable implementation of MPI standard for message-passing libraries.

orthogonal matrix
: A real square matrix A whose transpose and inverse are equal, that is, $A^T = A^{-1}$, and therefore $AA^T = A^TA = I$. All eigenvalues of an orthogonal matrix have the absolute value 1.

packed storage
: A storage scheme allowing you to store symmetric, Hermitian, or triangular matrices more compactly. The upper or lower triangle of a matrix is packed by columns in a one-dimensional array.

PDF
: Probability Density Function. The function that determines probability distribution for univariate or multivariate continuous random variable X. The probability density function $f(x)$ is closely related with the cumulative distribution function $F(x)$. For univariate distribution the relation is

$$F(x) = \int_{-\infty}^{x} f(t)dt .$$

For multivariate distribution the relation is

$$F(x_1,x_2, ..., x_n) = \int_{-\infty}^{x_1} \int_{-\infty}^{x_2} ... \int_{-\infty}^{x_n} f(t_1,t_2, ..., t_n)dt_1dt_2...dt_n .$$

positive-definite matrix	A square matrix *A* such that $Ax \cdot x > 0$ for any non-zero vector *x*. Here · denotes the dot product.
pseudorandom number generator	A completely deterministic algorithm that imitates truly random sequences.
QR factorization	Representation of an *m* by *n* matrix *A* as $A = QR$, where *Q* is an *m* by *m* orthogonal (unitary) matrix, and *R* is *n* by *n* upper triangular with real diagonal elements (if $m \geq n$) or trapezoidal (if $m < n$) matrix.
random stream	An abstract source of independent identically distributed random numbers of uniform distribution. In this manual a random stream points to a structure that uniquely defines a random number sequence generated by a basic generator associated with a given random stream.
RNG	Abbreviation for Random Number Generator. In this manual the term 'random number generators' stands for pseudorandom number generators, that is, generators based on completely deterministic algorithms imitating truly random sequences.
s	When found as the first letter of routine names, `s` indicates the usage of single-precision real data type.
ScaLAPACK	Stands for Scalable Linear Algebra PACKage.
Schur factorization	Representation of a square matrix *A* in the form $A = ZTZ^H$. Here *T* is an upper quasi-triangular matrix (for complex *A*, triangular matrix) called the Schur form of *A*; the matrix *Z* is orthogonal (for complex *A*, unitary). Columns of *Z* are called Schur vectors.

single precision	A floating-point data type. On Intel® processors, this data type allows you to store real numbers x such that $1.18*10^{-38} < \|x\| < 3.40*10^{38}$. For this data type, the machine precision (ε) is approximately 10^{-7}, which means that single-precision numbers usually contain no more than 7 significant decimal digits. For more information, refer to *Pentium® Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual*.
singular matrix	A matrix whose determinant is zero. If A is a singular matrix, the inverse A^{-1} does not exist, and the system of equations $Ax = b$ does not have a unique solution (that is, there exist no solutions or an infinite number of solutions).
singular value	The numbers defined for a given general matrix A as the eigenvalues of the matrix AA^H. *See also* SVD.
SMP	Abbreviation for Symmetric MultiProcessing. The MKL offers performance gains through parallelism provided by the SMP feature.
sparse BLAS	Routines performing basic vector operations on sparse vectors. Sparse BLAS routines take advantage of vectors' sparsity: they allow you to store only non-zero elements of vectors. *See* BLAS.
sparse vectors	Vectors in which most of the components are zeros.
storage scheme	The way of storing matrices. *See* full storage, packed storage, and band storage.
SVD	Abbreviation for Singular Value Decomposition. *See also* Singular value decomposition section in Chapter 5.
symmetric matrix	A square matrix A such that $a_{ij} = a_{ji}$.
transpose	The transpose of a given matrix A is a matrix A^T such that $(A^T)_{ij} = a_{ji}$ (rows of A become columns of A^T, and columns of A become rows of A^T).

trapezoidal matrix	A matrix A such that $A = (A_1A_2)$, where A_1 is an upper triangular matrix, A_2 is a rectangular matrix.
triangular matrix	A matrix A is called an upper (lower) triangular matrix if all its subdiagonal elements (superdiagonal elements) are zeros. Thus, for an upper triangular matrix $a_{ij} = 0$ when $i > j$; for a lower triangular matrix $a_{ij} = 0$ when $i < j$.
tridiagonal matrix	A matrix whose non-zero elements are in three diagonals only: the leading diagonal, the first subdiagonal, and the first super-diagonal.
unitary matrix	A complex square matrix A whose conjugate and inverse are equal, that is, that is, $A^H = A^{-1}$, and therefore $AA^H = A^HA = I$. All eigenvalues of a unitary matrix have the absolute value 1.
VML	Abbreviation for Vector Mathematical Library. *See* Chapter 9 of this book.
VSL	Abbreviation for Vector Statistical Library. *See* Chapter 10 of this book.
z	When found as the first letter of routine names, `z` indicates the usage of double-precision complex data type.

Bibliography

For more information about the BLAS, Sparse BLAS, LAPACK, ScaLAPACK, Sparse Solver, VML, VSL, and DFT routines, refer to the following publications:

- **BLAS Level 1**

  C. Lawson, R. Hanson, D. Kincaid, and F. Krough. *Basic Linear Algebra Subprograms for Fortran Usage*, ACM Transactions on Mathematical Software, Vol.5, No.3 (September 1979) 308-325.

- **BLAS Level 2**

  J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. *An Extended Set of Fortran Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, Vol.14, No.1 (March 1988) 1-32.

- **BLAS Level 3**

  J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling. *A Set of Level 3 Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software (December 1989).

- **Sparse BLAS**

  D. Dodson, R. Grimes, and J. Lewis. *Sparse Extensions to the FORTRAN Basic Linear Algebra Subprograms*, ACM Transactions on Math Software, Vol.17, No.2 (June 1991).

  D. Dodson, R. Grimes, and J. Lewis. *Algorithm 692: Model Implementation and Test Package for the Sparse Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, Vol.17, No.2 (June 1991).

- **LAPACK**

[LUG] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*, Third Edition, Society for Industrial and Applied Mathematics (SIAM), 1999.

[Golub96] G. Golub and C. Van Loan. *Matrix Computations*, Johns Hopkins University Press, Baltimore, third edition,1996.

- **ScaLAPACK**

[SLUG] L. Blackford, J. Choi, A.Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K.Stanley, D. Walker, and R. Whaley. *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics (SIAM), 1997.

- **Sparse Solver**

[1] I. S. Duff and J. Koster. *The design and use of algorithms for permuting large entries to the diagonal of sparse matrices*. SIAM J. Matrix Analysis and Applications, 20(4):889–901, 1999.

[2] G. Karypis and V. Kumar. *A fast and high quality multilevel scheme for partitioning irregular graphs*. SIAM Journal on Scientific Computing, 20(1):359–392, 1998.

[3] R. Menon L. Dagnum. *OpenMP: An Industry-Standard API for Shared-Memory Programming*. IEEE Computational Science & Engineering, 1:46–55, 1998. http://www.openmp.org.

[4] X.S. Li and J.W. Demmel. *A scalable sparse direct solver using static pivoting.* In Proceeding of the 9th SIAM conference on Parallel Processing for Scientific Computing, San Antonio, Texas, March 22-34,1999.

[5] J.W.H. Liu. *Modification of the Minimum-Degree algorithm by multiple elimination*. ACM Transactions on Mathematical Software, 11(2):141–153, 1985.

[6] O. Schenk. *Scalable Parallel Sparse LU Factorization Methods on Shared Memory Multiprocessors.* PhD thesis, ETH Zurich, 2000.

[7] O. Schenk and K. Gartner. *Two-level scheduling in PARDISO: Improved scalability on shared memory multiprocessing systems.* Parallel Computing, 28:187–197, 2002.

[8] O. Schenk and K. Gartner. *Solving unsymmetric sparse systems of linear equations with PARDISO*. Future Generation Computer Systems. Accepted, in press., 2003.

[9] O. Schenk and K. Gartner. *Sparse factorization with two-level scheduling in PARDISO.* In Proceeding of the 10th SIAM conference on Parallel Processing for Scientific Computing, Portsmouth, Virginia, March 12-14, 2001.

[10] O. Schenk, K. Gartner, and W. Fichtner. *Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors.* BIT, 40(1):158–176, 2000.

[11] P. Sonneveld. *CGS, a fast Lanczos-type solver for nonsymmetric linear systems.* SIAM Journal on Scientific and Statistical Computing, 10:36–52, 1989.

- **VSL**

[VSL Notes] Document included with Intel® MKL product (file name `vslnotes.pdf`).

[Bratley87] Bratley P., Fox B.L., and Schrage L.E. *A Guide to Simulation*. 2nd edition. Springer-Verlag, New York, 1987.

[Bratley88] Bratley P. and Fox B.L. *Implementing Sobol`s Quasirandom Sequence Generator*, ACM Transactions on Mathematical Software, Vol. 14, No. 1, Pages 88-100, March 1988.

[Bratley92] Bratley P., Fox B.L., and Niederreiter H. *Implementation and Tests of Low-Discrepancy Sequences*, ACM Transactions on Modeling and Computer Simulation, Vol. 2, No. 3, Pages 195-213, July 1992.

[Coddington94] Coddington, P. D. *Analysis of Random Number Generators Using Monte Carlo Simulation.* Int. J. Mod. Phys. C–5, 547, 1994.

[Gentle98] Gentle, James E. *Random Number Generation and Monte Carlo Methods*, Springer-Verlag New York, Inc., 1998.

[L'Ecuyer94] L'Ecuyer, Pierre. *Uniform Random Number Generation*. Annals of Operations Research, 53, 77–120, 1994.

[L'Ecuyer99] L'Ecuyer, Pierre. *Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure*. Mathematics of Computation, 68, 225, 249-260, 1999.

[L'Ecuyer99a] L'Ecuyer, Pierre. *Good Parameter Sets for Combined Multiple Recursive Random Number Generators.* Operations Research, 47, 1, 159-164, 1999.

[L'Ecuyer01] L'Ecuyer, Pierre. *Software for Uniform Random Number Generation: Distinguishing the Good and the Bad*. Proceedings of the 2001 Winter Simulation Conference, IEEE Press, 95–105, Dec. 2001.

[Kirkpatrick81] Kirkpatrick, S., and Stoll, E. *A Very Fast Shift-Register Sequence Random Number Generatory.* Journal of Computational Physics, V. 40. 517–526, 1981.

[Knuth81] Knuth, Donald E. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. 2nd edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.

[NAG] NAG Numerical Libraries. http://www.nag.co.uk/numeric/numerical_libraries.asp

[Sobol76] Sobol, I.M., and Levitan, Yu.L. *The production of points uniformly distributed in a multidimensional cube*. Preprint 40, Institute of Applied Mathematics, USSR Academy of Sciences, 1976 (In Russian).

- **DFT**

[1] E. Oran Brigham, *The Fast Fourier Transform and Its Applications*, Prentice Hall, New Jersey, 1988.

[2] Athanasios Papoulis, *The Fourier Integral and its Applications*, 2nd edition, McGraw-Hill, New York, 1984.

[3] Ping Tak Peter Tang, *DFTI, a New API for DFT: Motivation, Design, and Rationale*, July 2002.

[4] Charles Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM, Philadelphia, 1992

- **VML**

J.M.Muller. *Elementary functions: algorithms and implementation*, Birkhauser Boston, 1997.

IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-1985.

For a reference implementation of BLAS, sparse BLAS, LAPACK, and ScaLAPACK packages (without platform-specific optimizations) visit www.netlib.org.

Index

Symbols

A

B

E

F

G

H

M

N

S

T

U

V

W

X