Intel® Math Kernel Library

Reference Manual

Issued in U.S.A.
Document Number 630813-6001

Intel® Math Kernel Library Reference Manual

Document Number: 630813-6001

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

Revision	Revision History	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 Math Kernel library 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 Math Kernel Library release 2.1. Sparse BLAS section has been added in Chapter 2.	1/98
-007	Documents 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 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 Math Kernel Library release 4.0. Chapter 6 describing the VML functions has been added.	06/00
-011	Documents 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 Math Kernel Library release 6.0 beta. New DFT interface (chapter 8) and Vector Statistics Library functions (chapter 7) have been added.	07/02

Contents

Chapter 3 Fast Fourier Transforms

Chapter 4 LAPACK Routines: Linear Equations

Chapter 5 LAPACK Routines: Least Squares and Eigenvalue Problems

Chapter 8 Advanced DFT Interface

Overview 1

The Intel® Math Kernel Library (MKL) provides Fortran routines and functions that perform a wide variety of operations on vectors and matrices. The library also includes fast Fourier transform functions and new discrete Fourier transform interface, as well as vector mathematical and vector statistical functions with Fortran and C interfaces. The MKL enhances the performance of the programs that use it because the library has been optimized for Intel® processors.

This chapter introduces the Math Kernel Library and provides information about the organization of this manual.

About This Software

The 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)
- Fast Fourier transform routines (with Fortran and C interfaces)
- LAPACK routines for solving systems of linear equations
- LAPACK routines for solving least-squares problems, eigenvalue and singular value problems, and Sylvester's equations
- Vector Mathematical Library (VML) functions for computing core mathematical functions on vector arguments (with Fortran and C interfaces)
- Vector Statistics Library (VSL) functions for generating vectors of pseudorandom numbers with different types of statistical distributions
- Advanced Discrete Fourier Transform Interface (DFTI).

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

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.

Fast Fourier Transforms

Fast Fourier Transforms (FFTs) are used in digital signal processing and image processing and in partial differential equation (PDE) solvers. Combined with the BLAS routines, the FFTs contribute to the portability of the programs and provide a simplified interface between your program and the available library. To obtain more functionality and ease of use, consider also using the new DFT interface described in Chapter 8 .

LAPACK Routines

The Math Kernel Library covers the full set of the LAPACK computational and driver routines. These 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 4).

- Routines for solving least-squares problems, eigenvalue and singular value problems, and Sylvester's equations (see Chapter 5).

VML Functions

VML functions (see Chapter 6) 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 Statistics Library (VSL) functions (see Chapter 7) include a set of pseudorandom 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 Interface

The newly developed Discrete Fourier Transform interface (see Chapter 8) provides uniformity of DFT computation and combines functionality with ease of use. Both Fortran and C interface specification are given. Users are encouraged to migrate to the new interface in their application programs.

Performance Enhancements

The 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 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 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 *Release Notes*.

Platforms Supported

The 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 MKL includes a C-language interface for the fast Fourier transform functions, new discrete Fourier transform API, as well as for the Vector Mathematical Library and Vector Statistics Library functions.

About This Manual

This manual describes the routines of the Math Kernel Library. Each reference section describes a routine group 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 for each type of data with which each routine of the group is used. The following sections are also included:

Discussion	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 linear algebra and assumes a working knowledge of linear algebra and Fourier transform principles and vocabulary.

Manual Organization

The manual contains the following chapters and appendixes:

Chapter 1	Overview. Introduces the 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	Fast Fourier Transforms. Provides descriptions of fast Fourier transforms (FFT).
Chapter 4	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 5	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 6	Vector Mathematical Functions. Provides descriptions of VML functions for computing elementary mathematical functions on vector arguments.
Chapter 7	Vector Generators of Statistical Distributions. Provides descriptions of VSL functions for generating vectors of pseudorandom numbers.
Chapter 8	Advanced DFT Interface. Describes new interface for computing the Discrete Fourier Transform.
Appendix A	Routine and Function Arguments. Describes the major arguments of the BLAS routines and VML functions: vector and matrix arguments.
Appendix B	Code Examples. Provides code examples of calling BLAS functions and routines.

Appendix C CBLAS Interface to the BLAS. Provides the C interface to the BLAS.

The manual also includes a 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.

Related Publications

For more information about the BLAS, Sparse BLAS, LAPACK and VML 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 Mathematical 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

  E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Donagarra, 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.

  G. Golub and C. Van Loan. *Matrix Computations*, Johns Hopkins University Press, 1989.
- 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, and LAPACK packages (without platform-specific optimizations) visit www.netlib.org.

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 A.

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)
?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.

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

Discussion

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 = |Re*x*(1)| + |Im*x*(1)| + |Re*x*(2)| + |Im*x*(2)|+ ... + |Re*x*(*n*)| + |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.

```
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 )
```

Discussion

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.

```
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 )
```

Discussion

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.

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

Discussion

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.

?dotc

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

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

Discussion

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.

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

Discussion

The ?dotu functions perform a vector-vector reduction operation defined as $res = \sum(x * y)$

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

Input Parameters

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

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

Discussion

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.

```
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 )
```

Discussion

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.

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

Discussion

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.

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

Discussion

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.

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

Discussion

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.

```
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 )
```

Discussion

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 REAL for cscal DOUBLE PRECISION 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.

```
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 )
```

Discussion

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.

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

Discussion

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.

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

Discussion

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 |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 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

continued *

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

Routine Groups	Data Types	Description
?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
?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

```
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 )
```

Discussion

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 ≤ *kl*.

ku INTEGER. Specifies the number of super-diagonals of the matrix *a*. The value of *ku* must satisfy 0 ≤ *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

```
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 )
```

Discussion

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.

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

Discussion

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.

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

Discussion

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.

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

Discussion

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.

```
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 )
```

Discussion

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.

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

Discussion

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.

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

Discussion

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.

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

Discussion

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.

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

Discussion

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.

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

Discussion

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.

```
call chpr2 ( uplo, n, alpha, x, incx, y, incy, ap )
call zhpr2 ( uplo, n, alpha, x, incx, y, incy, ap )
```

Discussion

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.

```
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 )
```

Discussion

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.

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

Discussion

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.

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

Discussion

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.

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

Discussion

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.

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

Discussion

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.

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

Discussion

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.

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

Discussion

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.

```
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 )
```

Discussion

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.

```
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 )
```

Discussion

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*
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.

```
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 )
```

Discussion

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.

```
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 )
```

Discussion

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.

```
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 )
```

Discussion

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.

```
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 )
```

Discussion

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 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 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.

```
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 )
```

Discussion

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*')

transb 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.

```
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 )
```

Discussion

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.

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

Discussion

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.

```
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 )
```

Discussion

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.

```
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 )
```

Discussion

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.

```
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 )
```

Discussion

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.

```
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 )
```

Discussion

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*'**a*+*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).

```
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 )
```

Discussion

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).

```
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 )
```

Discussion

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*
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 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:

$x(1)=a(k_1), x(2)=a(k_2), \ldots x(nz)=a(k_{nz})$,

$indx(1)=k_1, indx(2)=k_2, \ldots indx(nz)=k_{nz}$.

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 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 |Re*x(i)*| + |Im*x(i)*|.
i?amin	index of the element with the smallest absolute value or, for complex flavors, the smallest sum |Re*x(i)*| + |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.

```
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 )
```

Discussion

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.

```
res = sdoti ( nz, x, indx, y )
res = ddoti ( nz, x, indx, y )
```

Discussion

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.

```
res = cdotci ( nz, x, indx, y )
res = zdotci ( nz, x, indx, y )
```

Discussion

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.

```
res = cdotui ( nz, x, indx, y )
res = zdotui ( nz, x, indx, y )
```

Discussion

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.

```
call sgthr ( nz, y, x, indx )
call dgthr ( nz, y, x, indx )
call cgthr ( nz, y, x, indx )
call zgthr ( nz, y, x, indx )
```

Discussion

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.

```
call sgthrz ( nz, y, x, indx )
call dgthrz ( nz, y, x, indx )
call cgthrz ( nz, y, x, indx )
call zgthrz ( nz, y, x, indx )
```

Discussion

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.

```
call sroti ( nz, x, indx, y, c, s )
call droti ( nz, x, indx, y, c, s )
```

Discussion

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.

```
call ssctr ( nz, x, indx, y )
call dsctr ( nz, x, indx, y )
call csctr ( nz, x, indx, y )
call zsctr ( nz, x, indx, y )
```

Discussion

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.

Fast Fourier Transforms 3

This chapter describes the fast Fourier transform (FFT) routines. The FFT routines included consist of two classes: one-dimensional and two-dimensional. Both one-dimensional and two-dimensional routines have been optimized to effectively use cache memory.

The chapter contains these 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 3-1 lists the one-dimensional FFT routine groups and the data types associated with them.

Table 3-1 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 3-1.

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)

```
call cfft1d ( r, n, isign, wsave )
call zfft1d ( r, n, isign, wsave )
```

Discussion

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).

```
void cfft1dc (float* r, float* i, int n, int isign, float* wsave)
void zfft1dc (double* r, double* i, int n, int isign, double* wsave)
```

Discussion

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 3-2 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 3-2 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).

```
call scfft1d ( r, n, isign, wsave )
call dzfft1d ( r, n, isign, wsave )
```

Discussion

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	RE z(2)	IM z(2)	...	RE z(n/2)	IM z(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).

```
void scfft1dc ( float* r, int n, int isign, float* wsave );
void dzfft1dc ( double* r, int n, int isign, double* wsave );
```

Discussion

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)	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.

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 $t_j = cmplx(r_j,0)$, 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 3-3 is identical to Table 3-2, 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 3-3 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).

```
call csfft1d ( r, n, isign, wsave )
call zdfft1d ( r, n, isign, wsave )
```

Discussion

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	RE*z*(2)	IM*z*(2)	...	RE*z*(*n*/2)	IM*z*(*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).

```
void csfft1dc ( float* r, int n, int isign, float* wsave )
void zdfft1dc ( double* r, int n, int isign, double* wsave )
```

Discussion

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 3-4 lists the two-dimensional FFT routine groups and the data types associated with them.

Table 3-4 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 3-4.

cfft2d/zfft2d

Fortran-interface routines. Compute the forward or inverse FFT of a complex matrix (in-place).

```
call cfft2d ( r, m, n, isign )
call zfft2d ( r, m, n, isign )
```

Discussion

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).

```
void cfft2dc ( float* r, float* i, int m, int n, int isign )
void zfft2dc ( double* r, double* i, int m, int n, int isign )
```

Discussion

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

$$z_{i,j} = \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} t_{k,l} * w_m^{-i*k} * w_n^{-j*l}, \quad 0 \le i \le m-1, \quad 0 \le j \le n-1$$

$t_{k,l} = cmplx(r_{k,l},0)$, where $r_{k,l}$ is a real input matrix, $0 \le k \le m\text{-}1$, $0 \le l \le n\text{-}1$. The mathematical result $z_{i,j}$, $0 \le i \le m\text{-}1$, $0 \le j \le n\text{-}1$, is the complex matrix of size (m,n). Each column is the complex conjugate-symmetric vector as follows:

for $0 \le j \le n\text{-}1$,

$z(m/2+i,j) = conjg(z(m/2-i,j))$, $1 \le i \le m/2\text{-}1$.
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 complex two-dimensional array of size $(m/2+1,n/2+1)$ or in the real two-dimensional array of size $(m+2,n+2)$. 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).

```
call scfft2d ( r, m, n )
call dzfft2d ( r, m, n )
```

Discussion

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 3-5 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 3-5 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 (see "Real-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 ≤ j ≤ n/2+1,

  z(1,n/2+1+j) = conjg(z(1,n/2+1-j)), 1 ≤ j ≤ 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 ≤ j ≤ n/2+1,

  z(m/2+1,n/2+1+j) = conjg(z(m/2+1,n/2+1-j)),
  1 ≤ j ≤ 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 ≤ i ≤ m/2-1, 1 ≤ j ≤ n .

- The rest matrix elements can be obtained from
  $z(m/2+1+i,j) = conjg(z(m/2+1-i,j))$,
  $1 \le i \le m/2-1$, $1 \le j \le n$.

The storage of the complex conjugate-symmetric matrix z for Fortran-interface is shown in Table 3-6.

Table 3-6 Fortran-interface Data Storage of CCS Format for the Real-to-Complex and Complex-to-Real Two-Dimensional FFTs

$z(1,1)$	0	$RE z(1,2)$	$IM z(1,2)$	...	$RE z(1,n/2)$	$IM z(1,n/2)$	$z(1,n/2+1)$	0
0	0	0	0	...	0	0	0	0
$RE z(2,1)$	$RE z(2,2)$	$RE z(2,3)$	$RE z(2,4)$	...	$RE z(2,n-1)$	$RE z(2,n)$	n/u	n/u
$IM z(2,1)$	$IM z(2,2)$	$IM z(2,3)$	$IM z(2,4)$	...	$IM z(2,n-1)$	$IM z(2,n)$	n/u	n/u
...	...	...	...	...	...	...	n/u	n/u
$RE z(m/2,1)$	$RE z(m/2,2)$	$RE z(m/2,3)$	$RE z(m/2,4)$	...	$RE z(m/2,n-1)$	$RE z(m/2,n)$	n/u	n/u
$IM z(m/2,1)$	$IM z(m/2,2)$	$IM z(m/2,3)$	$IM z(m/2,4)$	...	$IM z(m/2,n-1)$	$IM z(m/2,n)$	n/u	n/u
$z(m/2+1,1)$	0	$RE z(m/2+1,2)$	$IM z(m/2+1,2)$	...	$RE z(m/2+1,n/2)$	$IM z(m/2+1,n/2)$	$z(m/2+1,n/2+1)$	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).

```
void scfft2dc ( float* r, int m, int n )
void dzfft2dc ( double* r, int m, int n )
```

Discussion

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 3-7 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 3-7 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 (see "Real-to-Complex One-dimensional FFTs" above).

  The full complex vector *z*(i,0) is defined by:

  z(i,0) = cmplx(*r*(i,0), *r*(*m*/2+i+1,0)), 0 ≤ i ≤ *m*/2,

  z(*m*/2+i,0) = conjg(*z*(*m*/2-i,0)), 1 ≤ i ≤ *m*/2-1 .

  The full complex vector z(i,n/2) is defined by:

  z(i,*n*/2) = cmplx(*r*(i,*n*/2), *r*(*m*/2+i+1,*n*/2)), 0 ≤ i ≤ *m*/2,

  z(*m*/2+i,*n*/2) = conjg(*z*(*m*/2-i,*n*/2)), 1 ≤ i ≤ *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) = cmplx(*r*(i,j), *r*(i,*n*/2+1+j)),
  0 ≤ i ≤ *m*-1, 1 ≤ j ≤ *n*/2-1 .

- The rest matrix elements can be obtained from

  z(i,*n*/2+j) = conjg(*z*(i,*n*/2-j)),
  0 ≤ i ≤ *m*-1, 1 ≤ j ≤ *n*/2-1 .

The storage of the complex conjugate-symmetric matrix *z* for C-interface is shown in Table 3-8.

Table 3-8 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 $z_{i,j}$, $0 \le i \le m-1, \quad 0 \le j \le n-1$, 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 input can be stored in the complex two-dimensional array of size $(m/2+1, n/2+1)$ or in the real two-dimensional array of size $(m+2, n+2)$. For the details of data storage of CCS format see "Real-to-Complex One-dimensional FFTs" above.

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).

```
call csfft2d ( r, m, n )
call zdfft2d ( r, m, n )
```

Discussion

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 3-6.

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 3-5.

csfft2dc/zdfft2dc

C-interface routines.
Compute inverse FFT of a complex conjugate-symmetric matrix packed in CCS format (in-place).

```
void csfft2dc ( float* r, int m, int n );
void zdfft2dc ( double* r, int m, int n );
```

Discussion

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 3-8.

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 3-7 for the details.

LAPACK Routines: Linear Equations

4

This chapter describes the 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.

Routine Naming Conventions

For each routine introduced in this chapter, you can use the LAPACK name.

LAPACK names are listed in Tables 4-1 and 4-2, 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 A.

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.
$\|\|A\|\|_\infty = \max_i \sum_j \|a_{ij}\|$	The *infinity-norm* of the matrix A.
$\|\|A\|\|_1 = \max_j \sum_i \|a_{ij}\|$	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 4-1 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 4-2 lists similar routines for *complex* matrices.

Table 4-1 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 4-2 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.

```
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 )
```

Discussion

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.

```
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 )
```

Discussion

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* ≥ 0).
n	INTEGER. The number of columns in *A* (*n* ≥ 0).
kl	INTEGER. The number of sub-diagonals within the band of *A* (*kl* ≥ 0).
ku	INTEGER. The number of super-diagonals within the band of *A* (*ku* ≥ 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* ≥ 2*kl* + *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

$$|E| \le c(kl + ku + 1)\varepsilon P|L||U|$$

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.

```
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 )
```

Discussion

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.

```
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 )
```

Discussion

This routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite 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 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

$$|E| \le c(n)\varepsilon |U^H||U|, \quad |e_{ij}| \le c(n)\varepsilon \sqrt{a_{ii}a_{jj}}$$

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 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.

```
call spptrf ( uplo, n, ap, info )
call dpptrf ( uplo, n, ap, info )
call cpptrf ( uplo, n, ap, info )
call zpptrf ( uplo, n, ap, info )
```

Discussion

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 ($n \geq 0$).

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.

```
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 )
```

Discussion

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* ≥ *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

$$|E| \le c(kd+1)\varepsilon |U^H||U|, \quad |e_{ij}| \le c(kd+1)\varepsilon \sqrt{a_{ii}a_{jj}}$$

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.

```
call spttrf ( n, d, e, info )
call dpttrf ( n, d, e, info )
call cpttrf ( n, d, e, info )
call zpttrf ( n, d, e, info )
```

Discussion

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 sptrf, cpttrf DOUBLE PRECISION for dpttrf, zpttrf. Array, dimension (*n*). Contains the diagonal elements of A.
e	REAL for sptrf 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.

```
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 )
```

Discussion

This routine forms the Bunch-Kaufman factorization of a symmetric matrix:

if `uplo='U'`, $A = PUDU^TP^T$

if `uplo='L'`, $A = PLDL^TP^T$

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 $PUDU^TP^T$. If *uplo* = `'L'`, the array *a* stores the lower triangular part of the matrix A; A is factored as $PLDL^TP^T$.
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

$$|E| \leq c(n)\varepsilon P|U||D|\left|U^T\right|P^T$$

$c(n)$ 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.

```
call chetrf ( uplo, n, a, lda, ipiv, work, lwork, info )
call zhetrf ( uplo, n, a, lda, ipiv, work, lwork, info )
```

Discussion

This routine forms the Bunch-Kaufman factorization of a Hermitian matrix:

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 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 $PUDU^HP^T$.
If `uplo` = `'L'`, the array `a` stores the lower triangular part of the matrix A; A is factored as $PLDL^HP^T$.

`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* ≥ *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.

```
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 )
```

Discussion

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 ($n \geq 0$).

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.

```
call chptrf ( uplo, n, ap, ipiv, info )
call zhptrf ( uplo, n, ap, ipiv, info )
```

Discussion

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 ($n \geq 0$).

`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

$$|E| \leq c(n)\varepsilon P|U||D||U^T|P^T$$

$c(n)$ 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.

```
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)
```

Discussion

This routine solves for X the following systems of linear equations:

$AX = B$ 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 ?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 $A^TX = B$ is solved for X.
If *trans* = 'C', then $A^HX = B$ 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

$$|E| \le c(n)\varepsilon P|L||U|$$

$c(n)$ 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:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(n)\ \mathrm{cond}(A, x)\varepsilon$$

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 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.

?gbtrs

Solves a system of linear equations with an LU-factored band matrix, with multiple right-hand sides.

```
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)
```

Discussion

This routine solves for X the following systems of linear equations:

$AX = B$	if `trans='N'`,
$A^TX = B$	if `trans='T'`,
$A^HX = B$	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* ≥ 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 *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

$$|E| \le c(kl + ku + 1)\varepsilon P|L||U|$$

$c(k)$ 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 $\mathrm{cond}(A,x)$ 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 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.

```
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)
```

Discussion

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` ≥ 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

$$|E| \leq c(n)\varepsilon P|L||U|$$

$c(n)$ 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:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n)\ \mathrm{cond}(A, x)\varepsilon$$

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 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.

```
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 )
```

Discussion

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` ≥ 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

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 $\mathrm{cond}(A,x)$ can be much smaller than $\kappa_\infty(A)$.
The approximate number of floating-point operations for one right-hand side vector *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.

```
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 )
```

Discussion

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* ≥ 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 $\mathrm{cond}(A,x)$ can be much smaller than $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector *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.

```
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)
```

Discussion

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* ≥ *kd* +1).
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

$$|E| \le c(kd+1)\varepsilon P|U^H||U| \quad \text{or} \quad |E| \le c(kd+1)\varepsilon P|L^H||L|$$

$c(k)$ 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 4*n***kd* for real flavors and 16*n***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`.

```
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)
```

Discussion

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 ($n \geq 0$).
nrhs	`INTEGER`. The number of right-hand sides, i.e., the number of columns of the matrix B ($nrhs \geq 0$).

d	REAL for sptrs, 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 sptrs 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* ≥ 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.

```
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)
```

Discussion

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 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 $A = PUDU^TP^T$.
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 ($n \geq 0$).

`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* ≥ 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

$$|E| \le c(n)\varepsilon P|U||D||U^T|P^T \quad \text{or} \quad |E| \le c(n)\varepsilon P|L||D||L^T|P^T$$

$c(n)$ 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 $\mathrm{cond}(A,x)$ 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.

```
call chetrs (uplo, n, nrhs, a, lda, ipiv, b, ldb, info)
call zhetrs (uplo, n, nrhs, a, lda, ipiv, b, ldb, info)
```

Discussion

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 ($n \geq 0$).

`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 ?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* ≥ 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

$$|E| \le c(n)\varepsilon P|U||D||U^H|P^T \quad \text{or} \quad |E| \le c(n)\varepsilon P|L||D||L^H|P^T$$

$c(n)$ 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:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(n)\ \text{cond}(A, x)\varepsilon$$

where $\text{cond}(A,x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \le \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A,x)$ 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.

```
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 )
```

Discussion

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* ≥ 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:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(n)\ \text{cond}(A, x)\varepsilon$$

where $\text{cond}(A,x) = \| |A^{-1}||A||x| \|_\infty / \|x\|_\infty \le \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A,x)$ 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.

```
call chptrs ( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zhptrs ( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

Discussion

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 ***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 $A = PUDU^HP^T$.
If `uplo` = `'L'`, the array `ap` stores the packed factor L of the factorization $A = PLDL^HP^T$.

`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 ?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

$$|E| \le c(n)\varepsilon P|U||D||U^H|P^T \quad \text{or} \quad |E| \le c(n)\varepsilon P|L||D||L^H|P^T$$

$c(n)$ 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:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(n)\ \text{cond}(A, x)\varepsilon$$

where $\text{cond}(A,x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \le \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A,x)$ 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.

```
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)
```

Discussion

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',
$A^TX = B$	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 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 $\mathrm{cond}(A,x)$ 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 ?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.

```
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)
```

Discussion

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 ($n \geq 0$).

nrhs INTEGER. The number of right-hand sides ($nrhs \geq 0$).

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* ≥ 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 $\mathrm{cond}(A,x)$ 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.

```
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)
```

Discussion

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* ≥ 0).

kd INTEGER. The number of super-diagonals or sub-diagonals in the matrix A (*kd* ≥ 0).

nrhs INTEGER. The number of right-hand sides (*nrhs* ≥ 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* ≥ *kd* + 1.
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 $2n*kd$ for real flavors and $8n*kd$ 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.

```
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 )
```

Discussion

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 Discussion).
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.

```
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)
```

Discussion

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 Discussion).
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.

```
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 )
```

Discussion

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 *Discussion*).
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.

```
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 )
```

Discussion

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 *Discussion*).

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.

```
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 )
```

Discussion

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 ($n \geq 0$).

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 *Discussion*).

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.

```
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)
```

Discussion

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 *Discussion*).

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.

```
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)
```

Discussion

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 $A = U^HDU$ computed by ?pttrf :

$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1$ (since A 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 *Discussion*).

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.

```
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)
```

Discussion

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 ($n \geq 0$).

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 *Discussion*).

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.

```
call checon (uplo, n, a, lda, ipiv, anorm, rcond, work, rwork, info)
call zhecon (uplo, n, a, lda, ipiv, anorm, rcond, work, rwork, info)
```

Discussion

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 ($n \geq 0$).

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.

```
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 )
```

Discussion

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.

```
call chpcon ( uplo, n, ap, ipiv, anorm, rcond, work, rwork, info )
call zhpcon ( uplo, n, ap, ipiv, anorm, rcond, work, rwork, info )
```

Discussion

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.

```
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)
```

Discussion

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.

```
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)
```

Discussion

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 ($n \geq 0$).

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.

```
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 )
```

Discussion

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* ≥ *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.

```
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)
```

Discussion

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 ($n \geq 0$).

nrhs INTEGER. The number of right-hand sides ($nrhs \geq 0$).

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* ≥ 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 ?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.

```
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)
```

Discussion

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 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 $A^TX = B$. If *trans* = 'C', the system has the form $A^HX = B$.
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* ≥ 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 ?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.

```
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)
```

Discussion

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* ($n \geq 0$).

nrhs INTEGER. The number of right-hand sides , i.e., the number of columns of the matrix B ($nrhs \geq 0$).

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.

```
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)
```

Discussion

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* ($n \geq 0$).
nrhs	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
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* ≥ 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*).
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.

```
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)
```

Discussion

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 ($n \geq 0$).

nrhs — INTEGER. The number of right-hand sides ($nrhs \geq 0$).

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.

```
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)
```

Discussion

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* ≥ *kd* + 1.

ldafb — INTEGER. The first dimension of *afb*; *ldafb* ≥ *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 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.

```
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)
```

Discussion

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* ($n \geq 0$).
nrhs	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
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* 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*; $ldb \geq \max(1, n)$.
ldx	INTEGER. The leading dimension of *x*; $ldx \geq \max(1, n)$.

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.

```
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 ($n \geq 0$).

nrhs — INTEGER. The number of right-hand sides ($nrhs \geq 0$).

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.

```
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)
```

Discussion

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* ≥ 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 *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 ssyrfs / dsyrfs.

?sprfs

Refines the solution of a system of linear equations with a packed symmetric matrix and estimates the solution error.

```
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)
```

Discussion

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.

```
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)
```

Discussion

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 ($nrhs \ge 0$).

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.

```
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)
```

Discussion

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 $A^HX = B$.

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* ≥ 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*).
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.

```
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)
```

Discussion

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 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 $A^TX = B$.
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 *ap*.

n INTEGER. The order of the matrix *A* (*n* ≥ 0).

nrhs INTEGER. The number of right-hand sides (*nrhs* ≥ 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.

```
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)
```

Discussion

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* ≥ *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 $2n*kd$ floating-point operations for real flavors or $8n*kd$ 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.

```
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)
```

Discussion

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* ≥ 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* ≥ *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:

$$|XA - I| \le c(n)\varepsilon |X||P||L||U|$$

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.

```
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 ($n \geq 0$).

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

```
call spptri (uplo, n, ap, info)
call dpptri (uplo, n, ap, info)
call cpptri (uplo, n, ap, info)
call zpptri (uplo, n, ap, info)
```

Discussion

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:

$$\|XA - I\|_2 \le c(n)\varepsilon\kappa_2(A), \quad \|AX - I\|_2 \le c(n)\varepsilon\kappa_2(A)$$

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.

?sytri

Computes the inverse of a symmetric matrix.

```
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)
```

Discussion

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.

```
call chetri (uplo, n, a, lda, ipiv, work, info)
call zhetri (uplo, n, a, lda, ipiv, work, info)
```

Discussion

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.

```
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)
```

Discussion

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.

```
call chptri (uplo, n, ap, ipiv, work, info)
call zhptri (uplo, n, ap, ipiv, work, info)
```

Discussion

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 ($n \geq 0$).

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.

```
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)
```

Discussion

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:

$$|XA - I| \le c(n)\varepsilon|X||A|$$

$$|X - A^{-1}| \le c(n)\varepsilon|A^{-1}||A||X|$$

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.

```
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)
```

Discussion

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.

```
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)
```

Discussion

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* ≥0.

n — INTEGER. The number of columns of the matrix A, *n* ≥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 ≤ *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.

```
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)
```

Discussion

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* ≥0.
n	INTEGER. The number of columns of the matrix A, *n* ≥0.
kl	INTEGER. The number of sub-diagonals within the band of *A* (*kl* ≥ 0).
ku	INTEGER. The number of super-diagonals within the band of *A* (*ku* ≥ 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* ≥ *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 ≤ *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.

?poequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix and reduce its condition number.

```
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)
```

Discussion

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.

```
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)
```

Discussion

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* ($n \geq 0$).
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.

```
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)
```

Discussion

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 ($n \geq 0$).
kd	INTEGER. The number of super-diagonals or sub-diagonals in the matrix A ($kd \geq 0$).

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* ≥ *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* ≥ 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 4-3 lists the LAPACK driver routines for solving systems of linear equations with real or complex matrices.

Table 4-3 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.

```
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)
```

Discussion

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.

```
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)
```

Discussion

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': $\text{diag}(r)*A*\text{diag}(c)*\text{diag}(c)^{-1}*X = \text{diag}(r)*B$

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. 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* ($n \geq 0$).

nrhs — INTEGER. The number of right hand sides; the number of columns of the matrices *B* and *X* ($nrhs \geq 0$).

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* ≥ 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* $\neq$ '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 $0 < info \leq n$, 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 $i \leq n$, 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.

```
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)
```

Discussion

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 ith 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)
```

Discussion

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 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:
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*).

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* = diag(*r*)**A* *equed* = 'C': *A* = *A**diag(*c*) *equed* = 'B': *A* = diag(*r*)**A**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 $0 < info \le n$, 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.

?gtsv

Computes the solution to the system of linear equations with a tridiagonal matrix A and multiple right-hand sides.

```
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)
```

Discussion

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* ≥ 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.

```
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)
```

Discussion

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* ≥ 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` ≥ max(1, `n`).

`ldx` `INTEGER`. The first dimension of `x`; `ldx` ≥ 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* ≤ *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.

```
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)
```

Discussion

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 ($n \geq 0$).

nrhs — INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).

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* ≥ max(1, *n*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ 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.

```
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)
```

Discussion

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.

```
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)
```

Discussion

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* ≥ 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.

```
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)
```

Discussion

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 $\text{diag}(s)*A*\text{diag}(s)$ and B by $\text{diag}(s)*B$.

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.

```
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)
```

Discussion

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* ≥ 0).
nrhs	INTEGER. The number of right-hand sides; the number of columns in *B* (*nrhs* ≥ 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* ≥ *kd* +1)
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ 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.

```
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)
```

Discussion

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* ($n \geq 0$).
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; the number of columns in *B* ($nrhs \geq 0$).
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* ≥ *kd*+1.
ldafb	INTEGER. The first dimension of *afb*; *ldafb* ≥ *kd*+1.

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 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 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 $A=U^HU$ 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 *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.

```
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)
```

Discussion

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.

```
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)
```

Discussion

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.

```
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)
```

Discussion

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 ($nrhs \geq 0$).
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.

```
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)
```

Discussion

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 ($n \geq 0$).
nrhs	INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
a,af,b,work	REAL for 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 ith row and column of A was interchanged with the kth 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 mth 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 mth 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` $\leq$ `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.

```
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)
```

Discussion

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* ($n \geq 0$).
nrhs	INTEGER. The number of right-hand sides; the number of columns in *B* ($nrhs \geq 0$).
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* $\leq$ *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.

```
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)
```

Discussion

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 ($n \geq 0$).
nrhs	INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
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.

```
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)
```

Discussion

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* ≥ 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.

```
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)
```

Discussion

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 $i \leq n$, 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.

```
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)
```

Discussion

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* ≥ 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* $\leq$ *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.

```
call chpsv (uplo, n, nrhs, ap, ipiv, b, ldb, info)
call zhpsv (uplo, n, nrhs, ap, ipiv, b, ldb, info)
```

Discussion

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* ≥ 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

5

This chapter describes the 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 5-6).

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 5-7).

In the general case you may have a *rank-deficient least-squares problem*, with 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 5-74).

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 5-101).

Routines for solving eigenvalue problems with nonsymmetric or non-Hermitian matrices are described in the section *Nonsymmetric Eigenvalue Problems* (see page 5-174).

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 5-157).

* * *

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 4 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.

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 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 A.

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 5-2).
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: $\|x\|_E = \|x\|_2$).
$\theta(x, y)$	The *acute angle between vectors* x and y: $\cos \theta(x, y) = \|x \cdot y\| / (\|x\|_2 \|y\|_2)$.

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 5-1 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.

```
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 )
```

Discussion

The routine forms the *QR* factorization of a general *m* by *n* matrix *A* (see *Orthogonal Factorizations* on page 5-6). 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* ≥ *n*) See Application notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the factorization data as follows: If *m* ≥ *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.

```
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 )
```

Discussion

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 5-6). 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* ($m \geq 0$).
n	INTEGER. The number of columns in *A* ($n \geq 0$).
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* ≥ *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 $C = Q^T B$ (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).

?geqp3

Computes the QR factorization of a general m by n matrix with column pivoting using Level 3 BLAS.

```
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 )
```

Discussion

The routine forms the *QR* factorization of a general *m* by *n* matrix *A* with column pivoting: *AP* = *QR* (see *Orthogonal Factorizations* on page 5-6) 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* ≥ *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`.

```
call sorgqr ( m, n, k, a, lda, tau, work, lwork, info )
call dorgqr ( m, n, k, a, lda, tau, work, lwork, info )
```

Discussion

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 5-8) or `sgeqpf`/`dgeqpf` (see page 5-11). 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 $m \geq p$. 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* ($m \geq 0$).

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	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* ≥ *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`.

```
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 )
```

Discussion

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 5-8) or `sgeqpf`/`dgeqpf` (see page 5-11).

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` ≥ 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^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`.

```
call cungqr ( m, n, k, a, lda, tau, work, lwork, info )
call zungqr ( m, n, k, a, lda, tau, work, lwork, info )
```

Discussion

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 5-8) or `cgeqpf`/`zgeqpf` (see page 5-11). 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`.

```
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 )
```

Discussion

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 5-8) or `cgeqpf`/`zgeqpf` (see page 5-11).

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.

```
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 )
```

Discussion

The routine forms the *LQ* factorization of a general *m* by *n* matrix *A* (see *Orthogonal Factorizations* on page 5-6). 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* ≤ *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`.

```
call sorglq ( m, n, k, a, lda, tau, work, lwork, info )
call dorglq ( m, n, k, a, lda, tau, work, lwork, info )
```

Discussion

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 5-25). 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`.

```
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 )
```

Discussion

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 5-25).

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 `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`.

```
call cunglq ( m, n, k, a, lda, tau, work, lwork, info )
call zunglq ( m, n, k, a, lda, tau, work, lwork, info )
```

Discussion

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 5-25). 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`.

```
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 )
```

Discussion

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 5-25).

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 `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.

```
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 )
```

Discussion

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* ($m \geq 0$).
n	INTEGER. The number of columns in *A* ($n \geq 0$).
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* ≥ *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*; 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`.

```
call sorgql ( m, n, k, a, lda, tau, work, lwork, info )
call dorgql ( m, n, k, a, lda, tau, work, lwork, info )
```

Discussion

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*: $Q = H_k \cdots H_2 H_1$ 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* ≥ 0).
n	`INTEGER`. The number of columns of the matrix *Q* (*m* ≥ *n* ≥ 0).
k	`INTEGER`. The number of elementary reflectors whose product defines the matrix *Q* (*n* ≥ *k* ≥ 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`.

```
call cungql ( m, n, k, a, lda, tau, work, lwork, info )
call zungql ( m, n, k, a, lda, tau, work, lwork, info )
```

Discussion

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* ≥ 0).
n	`INTEGER`. The number of columns of the matrix *Q* (*m* ≥ *n* ≥ 0).
k	`INTEGER`. The number of elementary reflectors whose product defines the matrix *Q* (*n* ≥ *k* ≥ 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`.

```
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 )
```

Discussion

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` ≥ 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^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`.

```
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 )
```

Discussion

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 ith 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 ith 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.

```
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 )
```

Discussion

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* ≤ *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;* 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.

```
call sorgrq ( m, n, k, a, lda, tau, work, lwork, info )
call dorgrq ( m, n, k, a, lda, tau, work, lwork, info )
```

Discussion

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.

```
call cungrq ( m, n, k, a, lda, tau, work, lwork, info )
call zungrq ( m, n, k, a, lda, tau, work, lwork, info )
```

Discussion

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`.

```
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 )
```

Discussion

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* ($m \geq 0$).

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 `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`.

```
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 )
```

Discussion

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 ($m \geq 0$).

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 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.

```
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 )
```

Discussion

This routine reduces the *m*-by-*n* (*m* ≤ *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* ($m \geq 0$).
n	INTEGER. The number of columns in *A* ($n \geq m$).
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`.

```
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 )
```

Discussion

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 ith 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 ≥ 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 ?unmrz.

?unmrz

Multiplies a complex matrix by the unitary matrix defined from the factorization formed by `?tzrzf`.

```
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 )
```

Discussion

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 ($m \geq 0$).

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`	`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.

```
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)
```

Discussion

The routine forms the generalized *QR* factorization of an *n*-by-*m* matrix *A* and an *n*-by-*p* matrix *B* as $A = Q\,R$, $B = Q\,T\,Z$, 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:

$$R = \begin{array}{c} \\ m \\ n-m \end{array} \begin{array}{c} m \\ \begin{pmatrix} R_{11} \\ 0 \end{pmatrix} \end{array}, \quad \text{if } n \geq m$$

or

$$R = \begin{array}{c} n \\ n \end{array} \begin{array}{c} m-n \\ (R_{11} \quad R_{12}) \end{array}, \quad \text{if } n < m,$$

where R_{11} is upper triangular, and

$$T = \begin{array}{c} p-n \quad n \\ n \end{array} \begin{array}{c} \\ (0 \quad T_{12}) \end{array}, \quad \text{if } n \leq p, \text{ or}$$

$$T = \begin{array}{c} p \\ n-p \\ p \end{array} \begin{array}{c} \\ \begin{pmatrix} T_{11} \\ T_{21} \end{pmatrix} \end{array}, \quad \text{if } n > p$$

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.

```
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)
```

Discussion

The routine forms the generalized *RQ* factorization of an *m*-by-*n* matrix *A* and an *p*-by-*n* matrix *B* as $A = R\,Q$, $B = Z\,T\,Q$, 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:

$$R = \begin{array}{c} \\ m \end{array} \begin{array}{c} n-m \quad m \\ (\,0 \quad R_{12}\,) \end{array}, \quad \text{if } m \le n,$$

or

$$R = \begin{array}{c} \\ m-n \\ n \end{array} \begin{array}{c} n \\ \begin{pmatrix} R_{11} \\ R_{21} \end{pmatrix} \end{array}, \quad \text{if } m > n$$

where R_{11} or R_{21} is upper triangular, and

$$T = \begin{array}{c} \\ n \\ p-n \end{array} \begin{array}{c} n \\ \begin{pmatrix} T_{11} \\ 0 \end{pmatrix} \end{array}, \quad \text{if } p \ge n$$

or

$$T = \begin{array}{c} \\ p \end{array} \begin{array}{c} p \quad n-p \\ (\,T_{11} \quad T_{12}\,) \end{array}, \quad \text{if } p < n,$$

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 $m \leq n$, 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* ≥ *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:

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_{\min(m,\, n)} \geq 0.$$

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 *i*th 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 5-2 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 5-1 Decision Tree: Singular Value Decomposition

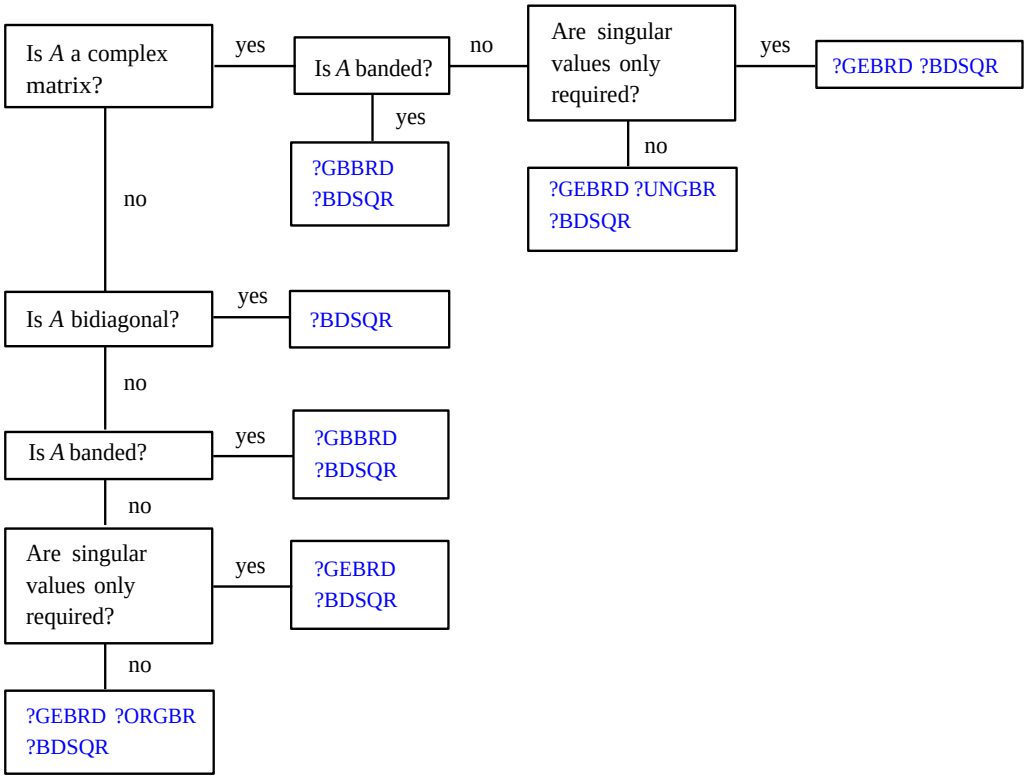


Figure 5-1 presents a decision tree that helps you choose the right sequence of routines for SVD, depending on whether you need singular values only or singular vectors as well, whether A is real or complex, and so on.

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 $U^H b = U_1^H Q^H b$.

?gebrd

Reduces a general matrix to bidiagonal form.

```
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 )
```

Discussion

The routine reduces a general *m* by *n* matrix *A* to a bidiagonal matrix *B* by an orthogonal (unitary) transformation.

If $m \geq n$, 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 ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
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 $m \geq n$, 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.

```
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 )
```

Discussion

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* ≥ *kl* + *ku* + 1).
ldq	INTEGER. The first dimension of the output array *q*. *ldq* ≥ max(1, *m*) if *vect* = 'Q' or 'B', *ldq* ≥ 1 otherwise.
ldpt	INTEGER. The first dimension of the output array *pt*. *ldpt* ≥ max(1, *n*) if *vect* = 'P' or 'B', *ldpt* ≥ 1 otherwise.
ldc	INTEGER. The first dimension of the array *c*. *ldc* ≥ max(1, *m*) if *ncc* > 0; *ldc* ≥ 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`.

```
call sorgbr ( vect, m, n, k, a, lda, tau, work, lwork, info )
call dorgbr ( vect, m, n, k, a, lda, tau, work, lwork, info )
```

Discussion

The routine generates the whole or part of the orthogonal matrices Q and P^T formed by the routines `sgebrd`/`dgebrd` (see page 5-76). 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 *Discussion* 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`.

```
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)
```

Discussion

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 5-76). 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: $m \geq 0$, $n \geq 0$, $k \geq 0$.

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 $m \geq k$;
2**m***k*(2**n* - *k*)	if *side* = 'R' and $n \geq k$;
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.

```
call cungbr ( vect, m, n, k, a, lda, tau, work, lwork, info )
call zungbr ( vect, m, n, k, a, lda, tau, work, lwork, info )
```

Discussion

The routine generates the whole or part of the unitary matrices Q and P^H formed by the routines cgebrd/zgebrd (see page 5-76). 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: $m \geq 0$, $n \geq 0$, $k \geq 0$.
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 — 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 *Discussion* 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.

```
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)
```

Discussion

Given an arbitrary complex matrix C, this routine forms one of the matrix products QC, Q^HC, CQ, CQ^H, PC, P^HC, CP, or CP^H, where Q and P are orthogonal matrices computed by a call to cgebrd/zgebrd (see page 5-76). 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: $m \geq 0$, $n \geq 0$, $k \geq 0$.

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.

```
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 )
```

Discussion

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 ($ncvt \geq 0$). 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 ($nru \geq 0$). 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 $Q^H C$ ($ncc \geq 0$). 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

$$|s_i - \sigma_i| \leq p(m, n)\varepsilon\sigma_i$$

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:

$$\theta(u_i, w_i) \leq p(m, n)\varepsilon / \min_{i \neq j}(|\sigma_i - \sigma_j|/|\sigma_i + \sigma_j|).$$

Here $\min_{i \neq j}(|\sigma_i - \sigma_j|/|\sigma_i + \sigma_j|)$ 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.

```
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 )
```

Discussion

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

$$Az = \lambda z. \text{ (or, equivalently, } z^H A = \lambda z^H).$$

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 5-3.

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 5-2 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 5-3.

Figure 5-2 Decision Tree: Real Symmetric Eigenvalue Problems

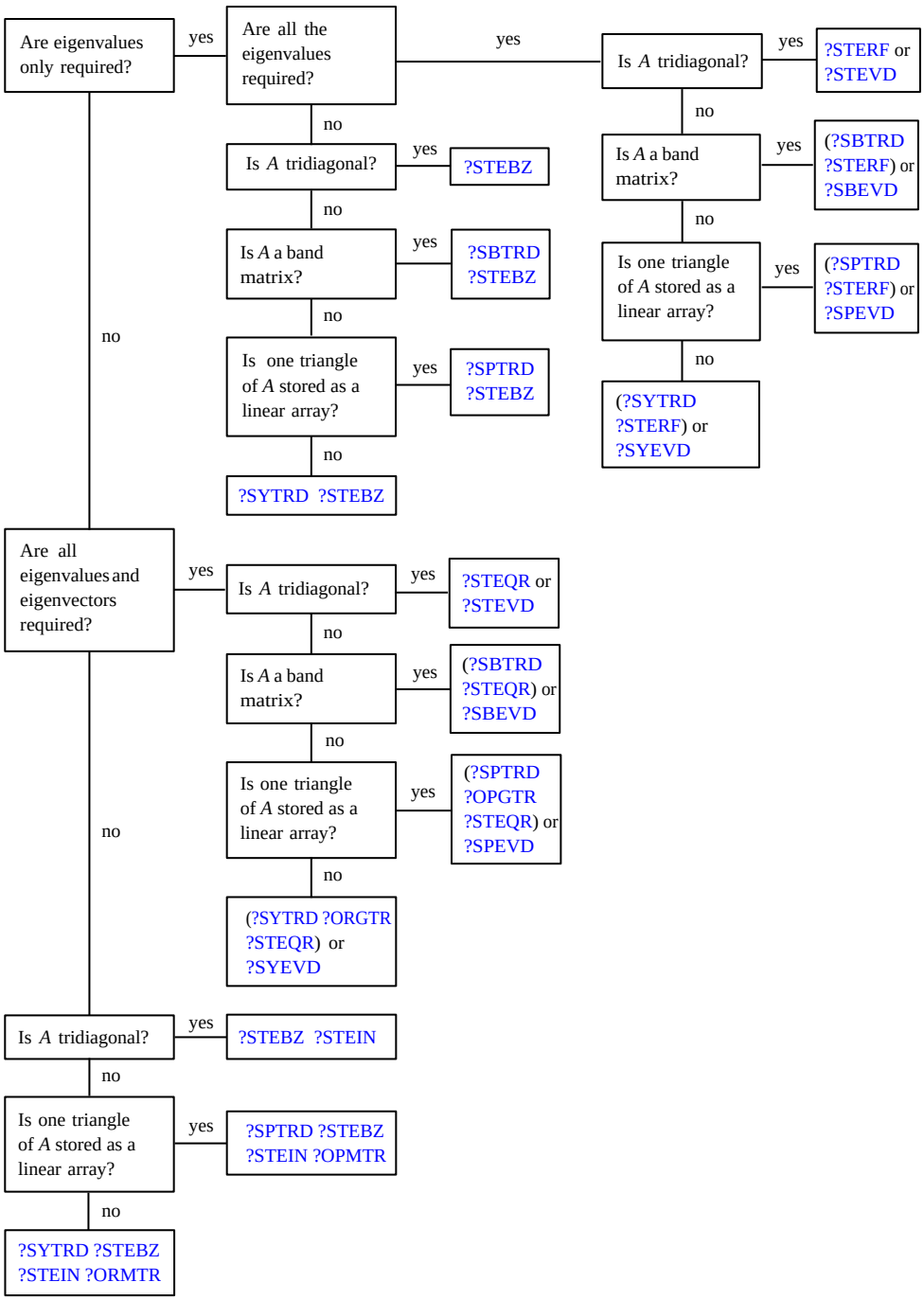


Figure 5-3 Decision Tree: Complex Hermitian Eigenvalue Problems

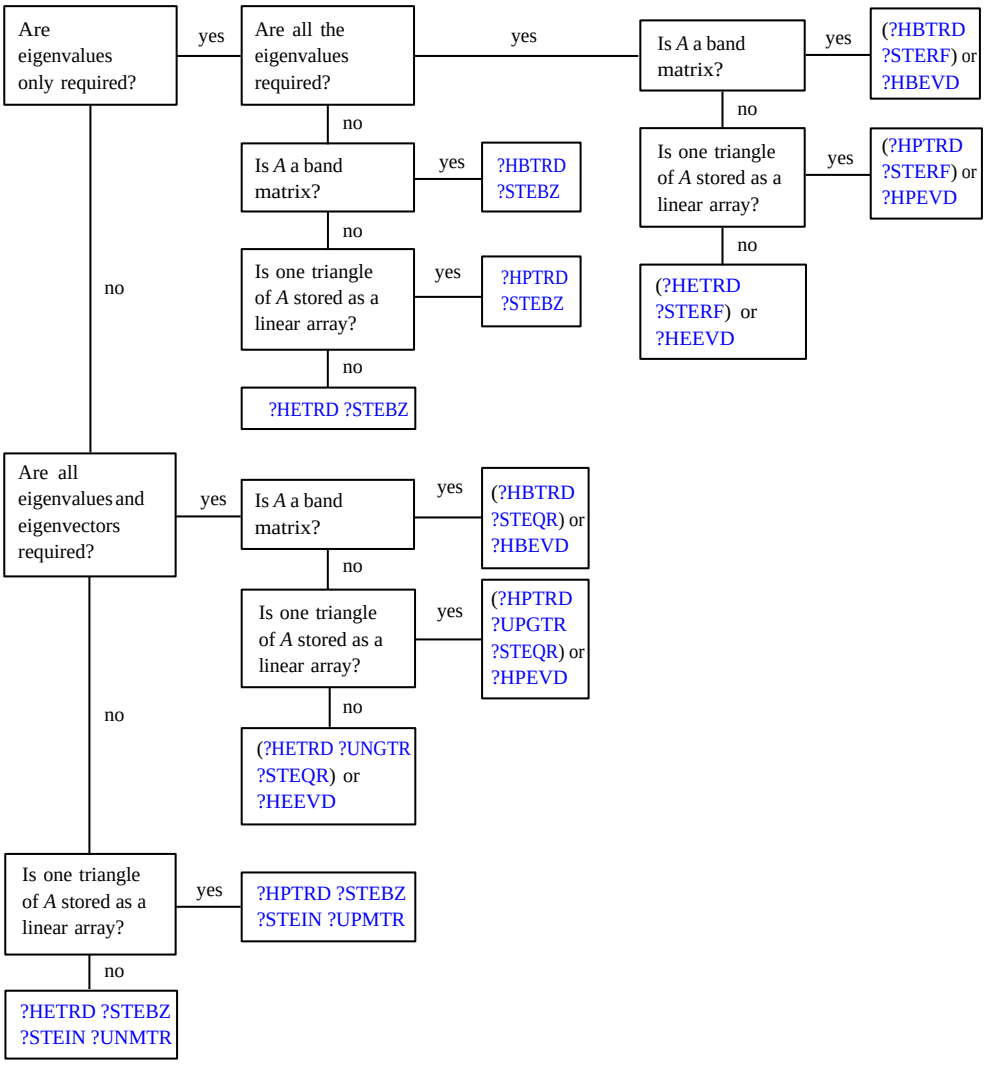


Table 5-3 Computational Routines for Solving Symmetric Eigenvalue Problems

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.

```
call ssytrd ( uplo,n,a,lda,d,e,tau,work,lwork,info )
call dsytrd ( uplo,n,a,lda,d,e,tau,work,lwork,info )
```

Discussion

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.

```
call sorgtr ( uplo, n, a, lda, tau, work, lwork, info )
call dorgtr ( uplo, n, a, lda, tau, work, lwork, info )
```

Discussion

The routine explicitly generates the *n* by *n* orthogonal matrix *Q* formed by ?sytrd (see page 5-105) 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* ($n \geq 0$).
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 ($lwork \geq n$) 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`.

```
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 )
```

Discussion

The routine multiplies a real matrix C by Q or Q^T, where Q is the orthogonal matrix Q formed by `?sytrd` (see page 5-105) 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 ($m \geq 0$).
n	INTEGER. The number of columns in C ($n \geq 0$).
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^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* 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.

```
call chetrd ( uplo,n,a,lda,d,e,tau,work,lwork,info )
call zhetrd ( uplo,n,a,lda,d,e,tau,work,lwork,info )
```

Discussion

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* ≥ *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.

```
call cungtr ( uplo, n, a, lda, tau, work, lwork, info )
call zungtr ( uplo, n, a, lda, tau, work, lwork, info )
```

Discussion

The routine explicitly generates the *n* by *n* unitary matrix *Q* formed by ?hetrd (see page 5-111) 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* ≥ *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`.

```
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 )
```

Discussion

The routine multiplies a complex matrix C by Q or Q^H, where Q is the unitary matrix Q formed by `?hetrd` (see page 5-111) 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 ($m \geq 0$).

n — INTEGER. The number of columns in C ($n \geq 0$).

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^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* (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.

```
call ssptrd ( uplo,n,ap,d,e,tau,info )
call dsptrd ( uplo,n,ap,d,e,tau,info )
```

Discussion

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.

```
call sopgtr ( uplo, n, ap, tau, q, ldq, work, info )
call dopgtr ( uplo, n, ap, tau, q, ldq, work, info )
```

Discussion

The routine explicitly generates the *n* by *n* orthogonal matrix *Q* formed by ?sptrd (see page 5-117) 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* ($n \geq 0$).

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.

```
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)
```

Discussion

The routine multiplies a real matrix C by Q or Q^T, where Q is the orthogonal matrix Q formed by ?sptrd (see page 5-117) 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.

```
call chptrd ( uplo,n,ap,d,e,tau,info )
call zhptrd ( uplo,n,ap,d,e,tau,info )
```

Discussion

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.

```
call cupgtr ( uplo, n, ap, tau, q, ldq, work, info )
call zupgtr ( uplo, n, ap, tau, q, ldq, work, info )
```

Discussion

The routine explicitly generates the *n* by *n* unitary matrix *Q* formed by ?hptrd (see page 5-122) 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.

```
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)
```

Discussion

The routine multiplies a complex matrix *C* by *Q* or Q^H, where *Q* is the unitary matrix *Q* formed by ?hptrd (see page 5-122) 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* ≥ 0).

n — INTEGER. The number of columns in *C* (*n* ≥ 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.

```
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)
```

Discussion

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 ($n \geq 0$).
kd	INTEGER. The number of super- or sub-diagonals in A ($kd \geq 0$).
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 ≥ max(1, *n*) if *vect* = 'V';
ldq ≥ 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.

```
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)
```

Discussion

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 ≥ max(1, *n*) if *vect* = 'V';
ldq ≥ 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.

```
call ssterf ( n, d, e, info )
call dsterf ( n, d, e, info )
```

Discussion

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 5-134).

Input Parameters

`n`	`INTEGER`. The order of the matrix T ($n \geq 0$).
`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).

```
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 )
```

Discussion

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,

$$Tz_i = \lambda_i z_i \quad \text{for } i = 1, 2, \ldots, n.$$

(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 5-132). If T is positive-definite, `?pteqr` (page 5-146) 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.

```
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)
```

Discussion

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 $n \leq 1$ 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 $2^k \geq n$.

If *compz* = 'I' and $n > 1$ then *lwork* must be at least $(1 + 4n + n^2)$.

If *compz* = 'N' or $n \leq 1$ 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 $n \leq 1$, *lwork* must be at least 1.
If *compz* = 'V' and $n > 1$, *lwork* must be at least n^2.

If *compz* = 'N' or $n \leq 1$, *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 $2^k \geq n$.

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.

```
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)
```

Discussion

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 ≥ 1 if *jobz* = 'N';
ldz ≥ max(1, *n*) if *jobz* = 'V'.

lwork — INTEGER. The dimension of the array *work*,
lwork ≥ max(1, 18*n*).

iwork — INTEGER.
Workspace array, DIMENSION (*liwork*).

liwork — INTEGER. The dimension of the array *iwork*,
lwork ≥ max(1, 10*n*).

Output Parameters

d, e — On exit, *d* and *e* are overwritten.

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 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.

```
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 )
```

Discussion

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,

$$Tz_i = \lambda_i z_i \quad \text{for } i = 1, 2, \ldots, n.$$

(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* ≥ 1 if *compz* = 'N'; *ldz* ≥ 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.

```
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)
```

Discussion

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 $T = \mathrm{diag}(T_1, T_2, ...)$. 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.

```
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 )
```

Discussion

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 5-149). 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.

```
call sdisna (job, m, n, d, sep, info)
call ddisna (job, m, n, d, sep, info)
```

Discussion

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* ≥ 0).

n — `INTEGER`. If *job* = `'L'`, or `'R'`, the number of columns of the matrix (*n* ≥ 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 5-101).

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 5-4 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.

```
call ssygst ( itype, uplo, n, a, lda, b, ldb, info )
call dsygst ( itype, uplo, n, a, lda, b, ldb, info )
```

Discussion

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: $B = U^T U$ or $B = LL^T$ (see page 4-14).

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'`, 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*: $B = U^TU$ 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.

```
call chegst ( itype, uplo, n, a, lda, b, ldb, info )
call zhegst ( itype, uplo, n, a, lda, b, ldb, info )
```

Discussion

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 4-14).

Input Parameters

itype — `INTEGER`. Must be 1 or 2 or 3.
If *itype* = 1, the generalized eigenproblem is $Az = \lambda Bz$;
for *uplo* = `'U'`: $C = U^{-H} A U^{-1}$, $z = U^{-1} y$;
for *uplo* = `'L'`: $C = L^{-1} A L^{-H}$, $z = L^{-H} y$.
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 ($n \geq 0$).
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.

```
call sspgst ( itype, uplo, n, ap, bp, info )
call dspgst ( itype, uplo, n, ap, bp, info )
```

Discussion

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 $Cy = \lambda y$, 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 4-16).

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.

```
call chpgst ( itype, uplo, n, ap, bp, info )
call zhpgst ( itype, uplo, n, ap, bp, info )
```

Discussion

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 $Cy = \lambda y$, 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: $B = U^H U$ or $B = LL^H$ (see page 4-16).

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', *ap* stores the packed upper triangle of A; you must supply B in the factored form $B = U^H U$.
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 ($n \geq 0$).

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).

*b*p(*) 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`.

```
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 )
```

Discussion

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* ≥ *kb* ≥ 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`.

```
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 )
```

Discussion

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 ($n \geq 0$).
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* ≥ *kb* ≥ 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` .

```
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 )
```

Discussion

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 5-5 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 5-4 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 5-5.

Table 5-5 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 5-4 Decision Tree: Real Nonsymmetric Eigenvalue Problems

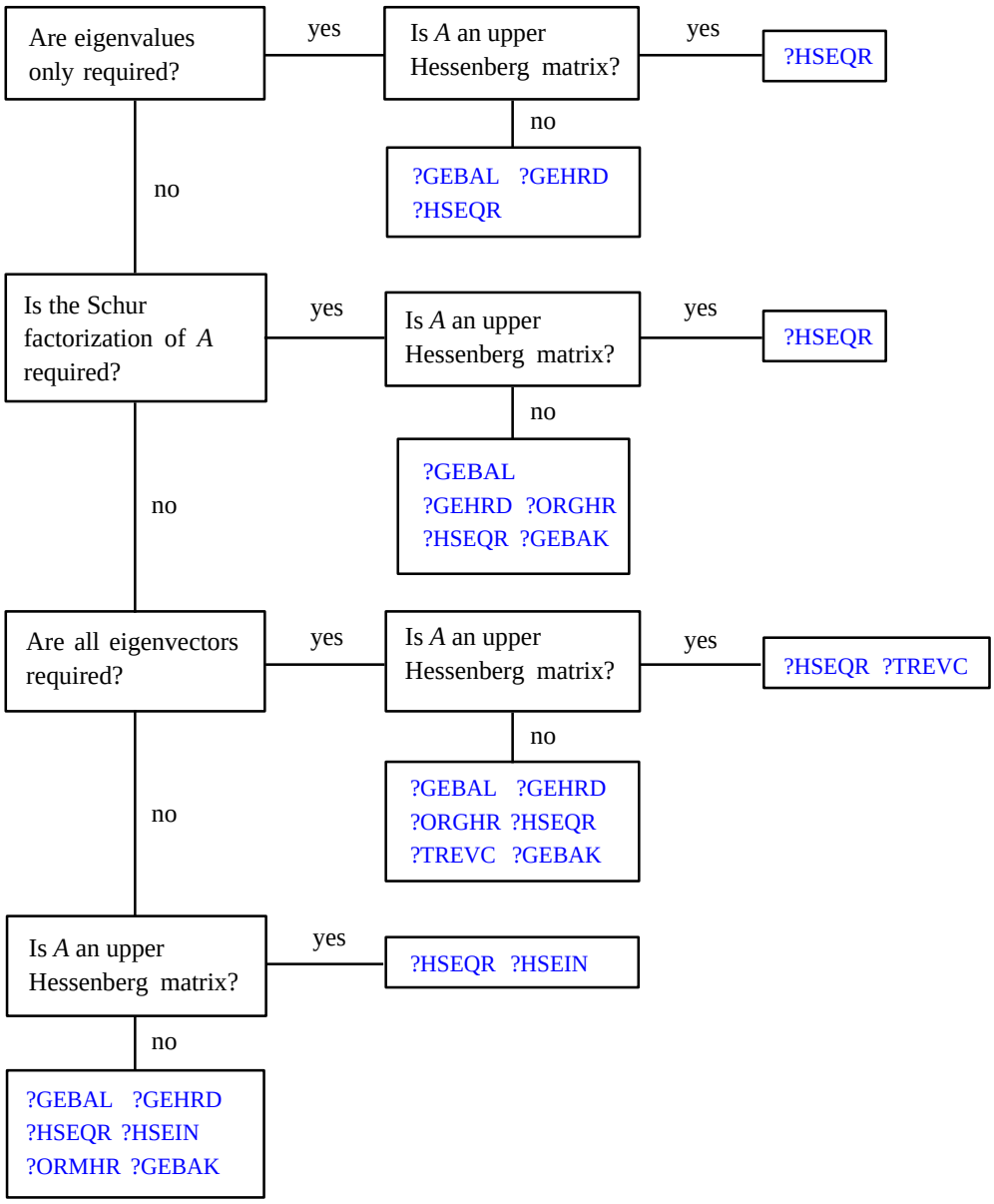
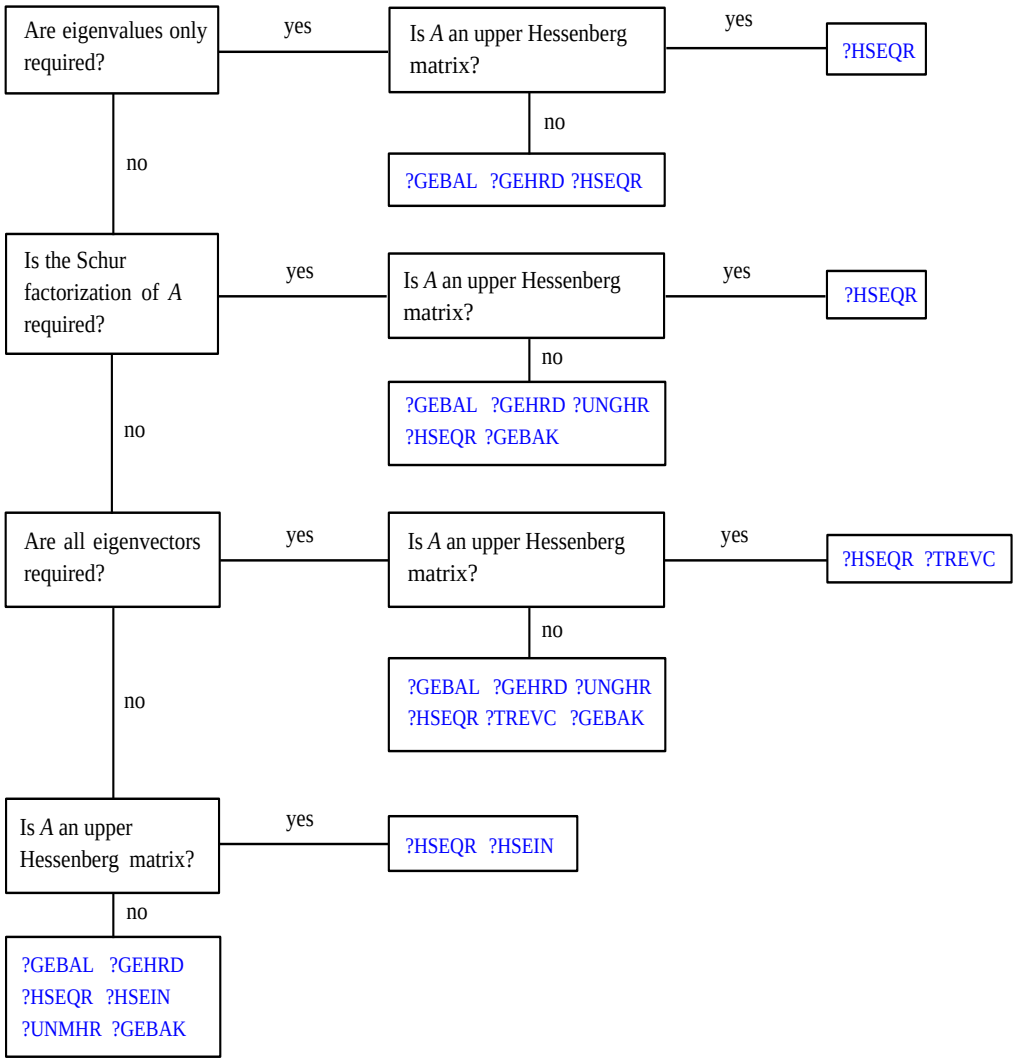


Figure 5-5 Decision Tree: Complex Non-Hermitian Eigenvalue Problems



?gehrd

Reduces a general matrix to upper Hessenberg form.

```
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 )
```

Discussion

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 ($n \geq 0$).
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 $1 \leq ilo \leq ihi \leq n$; 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, *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 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.

```
call sorghr ( n, ilo, ihi, a, lda, tau, work, lwork, info )
call dorghr ( n, ilo, ihi, a, lda, tau, work, lwork, info )
```

Discussion

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.

```
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 )
```

Discussion

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* ≥ 0).
n	INTEGER. The number of columns in *C* (*n* ≥ 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 ≤ *ilo* ≤ *ihi* ≤ *m*. If *m* = 0 and *side* = 'L', then *ilo* = 1 and *ihi* = 0. If *n* > 0 and *side* = 'R', then 1 ≤ *ilo* ≤ *ihi* ≤ *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.

```
call cunghr ( n, ilo, ihi, a, lda, tau, work, lwork, info )
call zunghr ( n, ilo, ihi, a, lda, tau, work, lwork, info )
```

Discussion

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 ($n \geq 0$).
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 $n = 0$, 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.

```
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 )
```

Discussion

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 ($m \geq 0$).

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 ≤ *ilo* ≤ *ihi* ≤ *m*. If *m* = 0 and *side* = 'L', then *ilo* = 1 and *ihi* = 0. If *n* > 0 and *side* = 'R', then 1 ≤ *ilo* ≤ *ihi* ≤ *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* ≥ 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^HC 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.

```
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 )
```

Discussion

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 5-193) 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.

```
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 )
```

Discussion

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 5-190). 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 ($n \geq 0$).
ilo, ihi	INTEGER. The values *ilo* and *ihi*, as returned by ?gebal. (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; 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.

```
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)
```

Discussion

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 5-178), call `?orghr` to form Q explicitly (page 5-180) and then pass Q to `?hseqr` with *compz* = `'V'`.

You can also call `?gebal` (page 5-190) 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 5-193) 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` This parameter is currently redundant.

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.
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 5-210).

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.

```
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 )
```

Discussion

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 5-182) or `?unmhr` (page 5-187).

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 5-195); 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 ≤ *mm* ≤ *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.

```
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 )
```

Discussion

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: $A = Q\,T\,Q^H$, as computed by ?hseqr (see page 5-195).

The right eigenvector x and the left eigenvector y of T corresponding to an eigenvalue w, are defined by:

$$T\,x = w\,x \quad , \qquad y^H T = w\,y^H$$

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 $Q\,X$ and/or $Q\,Y$, where Q is an input matrix.
If Q is the orthogonal/unitary factor that reduces a matrix A to Schur form T, then $Q\,X$ and $Q\,Y$ 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 Shur 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: $\theta(y_i,x_i) \leq (c(n)\varepsilon\|T\|_2)/\text{sep}_i$ 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.

```
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 )
```

Discussion

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 5-215) 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 ($n \geq 0$).

`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.

```
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 )
```

Discussion

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.

```
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)
```

Discussion

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 $AQ_1 = Q_1T_{11}$, 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 ($n \geq 0$).

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 ≤ *m* ≤ *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.

```
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 )
```

Discussion

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 $\alpha_i \pm \beta_i \neq 0$, 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 $\mathrm{op}(B) = B$.
If *tranb* = 'T', then $\mathrm{op}(B) = B^T$ (real flavors only).
If *tranb* = 'C', then $\mathrm{op}(B) = B^H$.

isgn INTEGER. Indicates the form of the Sylvester equation.
If *isgn* = +1, $\mathrm{op}(A)X + X\mathrm{op}(B) = \alpha C$.
If *isgn* = –1, $\mathrm{op}(A)X - X\mathrm{op}(B) = \alpha C$.

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 5-6 lists LAPACK routines used to solve the generalized nonsymmetric eigenvalue problems and the generalized Sylvester equation.

Table 5-6 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 Shur 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 Shur 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 Shur canonical form.

?gghrd

Reduces a pair of matrices to generalized upper Hessenberg form using orthogonal/unitary transformations.

```
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 )
```

Discussion

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 ($n \geq 0$).

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 $1 \leq ilo \leq ihi \leq n$;
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 $Q_1 Q$.

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 $Z_1 Z$.
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.

```
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 )
```

Discussion

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* ($n \geq 0$).

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.

```
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)
```

Discussion

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 $1 \le ilo \le ihi \le n$; 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).

```
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 )
```

Discussion

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 Shur form.

compq — CHARACTER*1. Must be 'N', 'I', or 'V'.
If *compq* = 'N', left Shur vectors (*q*) are not computed;
If *compq* = 'I', *q* is initialized to the unit matrix and the matrix of left Shur 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 Shur vectors (*q*) are not computed;
If *compz* = 'I', *z* is initialized to the unit matrix and the matrix of right Shur 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* ≥ 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*).

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 Shur 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 Shur 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 Shur 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 Shur 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 Shur 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 Shur vectors of the pair (*H*,*T*), and if *compq* = 'V', *q* is overwritten by the orthogonal/unitary matrix of left Shur vectors of (*A*,*B*).

z	On exit, if *compz* = 'I', *z* is overwritten by the orthogonal/unitary matrix of right Shur vectors of the pair (*H*,*T*), and if *compz* = 'V', *z* is overwritten by the orthogonal/unitary matrix of right Shur 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 Shur 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 Shur 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.

```
call stgevc ( side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr,
              ldvr, mm, m, work, info )
call stgevc ( side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr,
              ldvr, mm, m, work, info )
call stgevc ( side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr,
              ldvr, mm, m, work, rwork, info )
call stgevc ( side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr,
              ldvr, mm, m, work, rwork, info )
```

Discussion

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:

$$S\,x = w\,P\,x \quad , \qquad y^H S = w\,y^H P$$

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 Shur 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* ≥ 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 Shur 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 Shur 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 Shur decomposition of a pair of matrices (A,B) so that one diagonal block of (A,B) moves to another row index.

```
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 )
```

Discussion

This routine reorders the generalized real-Schur/Shur 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/Shur 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 *wantq* = .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, 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* ≥ 1. If *wantq* = .TRUE., then *ldq* ≥ max(1,*n*).
ldz	INTEGER. The first dimension of *z*; If *wantz* = .FALSE., then *ldz* ≥ 1. If *wantz* = .TRUE., then *ldz* ≥ 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 Shur decomposition of a pair of matrices (A,B) so that a selected cluster of eigenvalues appears in the leading diagonal blocks of (A,B).

```
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 )
```

Discussion

This routine reorders the generalized real-Schur/Shur 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/Shur 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` ≥ 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, 2*m*(*n*–*m*)).
For complex flavors:
If *ijob* = 1, 2, or 4, *liwork* ≥ *n*+2.
If *ijob* = 3 or 5, *liwork* ≥ max(*n*+2, 2*m*(*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 Shur 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 ≤ *m* ≤ *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. $0 < pl, pr \leq 1$. 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 $ijob \geq 2$, $dif(1:2)$ 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.

```
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 )
```

Discussion

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/Shur 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` ≤ 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 Shur canonical form.

```
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 )
```

Discussion

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* ≥ 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` ≥ 1.
If `job` = `'E'` or `'B'`, then `ldvl` ≥ max(1,`n`).

`ldvr` — `INTEGER`. The first dimension of `vr`; `ldvr` ≥ 1.
If `job` = `'E'` or `'B'`, then `ldvr` ≥ max(1,`n`).

`mm` — `INTEGER`. The number of elements in the arrays `s` and `dif` (`mm` ≥ `m`).

`lwork` — `INTEGER`. The dimension of the array `work`.
For real flavors:
`lwork` ≥ `n`.
If `job` = `'V'` or `'B'`, `lwork` ≥ $2n(n+2)+16$.
For complex flavors:
`lwork` ≥ `1`.
If `job` = `'V'` or `'B'`, `lwork` ≥ $2n^2$.

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 5-7 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.

```
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 )
```

Discussion

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* ≥ 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* ≥ 0).

p — INTEGER. The number of rows of the matrix *B* (*p* ≥ 0).

n — INTEGER. The number of columns of the matrices *A* and *B* (*n* ≥ 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 *Discussion* section.
b	Overwritten by the triangular matrix described in the *Discussion* 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.

```
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 )
```

Discussion

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*:

$$A = \begin{array}{c} \\ k \\ l \\ m-k-l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \\ 0 & 0 & 0 \end{pmatrix} \end{array}, \quad \text{if } m-k-l \geq 0$$

$$= \begin{array}{c} \\ k \\ m-k \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \end{pmatrix} \end{array}, \quad \text{if } m-k-l < 0$$

$$B = \begin{array}{c} \\ l \\ p-l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & 0 & B_{13} \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

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

$C = \text{diag} (alpha(k+1),...,alpha(k+l))$
$S = \text{diag} (beta(k+1),...,beta(k+l))$
$C^2 + S^2 = I$
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, $\begin{pmatrix} R_{11} R_{12} R_{13} \\ 0 \quad R_{22} R_{23} \end{pmatrix}$ 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 ($m \geq 0$).
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 ($n \geq 0$).
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 5-8 lists routines described in more detail below.

Table 5-8 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.

```
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 )
```

Discussion

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
   minimize $\| b - A x \|_2$

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
   minimize $\| b - A^H x \|_2$

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* ≥ *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* ≥ *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* ≥ *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.

```
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 )
```

Discussion

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:

$$AP = Q \begin{pmatrix} R_{11} R_{12} \\ 0 \quad R_{22} \end{pmatrix}$$

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 ≥ max(*mn*+3*n*+1, 2**mn* + *nrhs*),

where *mn* = min(*m*, *n*).

The block algorithm requires that:

lwork ≥ max(*mn*+2*n*+*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 ≥ *mn* + max(2**mn*, *n*+1, *mn* + *nrhs*),

where *mn* = min(*m*, *n*).

The block algorithm requires that:

lwork ≥ *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.

```
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 )
```

Discussion

This routine computes the minimum norm solution to a real linear least squares problem:

minimize $\| b - A x \|_2$

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*) ≤ *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* ≥ *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.

```
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 )
```

Discussion

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* ≥ 0).
n	INTEGER. The number of columns of the matrix *A* (*n* ≥ 0).
nrhs	INTEGER. The number of right-hand sides; the number of columns in *B* (*nrhs* ≥ 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*) ≤ *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 $m \geq n$ 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 $m \geq n$,
$lwork \geq 12n + 2n*smlsiz + 8n*nlvl + n*nrhs + (smlsiz+1)^2$;

If $m < n$,
$lwork \geq 12m + 2m*smlsiz + 8m*nlvl + m*nrhs + (smlsiz+1)^2$;

For complex flavors:

If $m \geq n$,
$lwork \geq 2n + n*nrhs$;

If $m < n$,
$lwork \geq 2m + m*nrhs$;

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 $m \geq n$,
$lrwork \geq 10n + 2n*smlsiz + 8n*nlvl + 3*smlsiz*nrhs + (smlsiz+1)^2$;

If $m < n$,
$lrwork \geq 10m + 2m*smlsiz + 8m*nlvl + 3*smlsiz*nrhs + (smlsiz+1)^2$.

Generalized LLS Problems

This section describes LAPACK driver routines used for solving generalized linear least-squares problems. Table 5-9 lists routines described in more detail below.

Table 5-9 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.

```
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 )
```

Discussion

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

$$\text{rank}(B) = p \quad \text{and} \quad \text{rank} \begin{pmatrix} A \\ B \end{pmatrix} = n \ .$$

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` ≥ 0).
`n`	`INTEGER`. The number of columns of the matrices *A* and *B* (`n` ≥ 0).
`p`	`INTEGER`. The number of rows of the matrix *B* (0 ≤ `p` ≤ `n` ≤ `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` ≥ 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.

```
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 )
```

Discussion

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

$$lwork \geq m+\min(n, p)+\max(n, p)*nb,$$

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 5-10 lists routines described in more detail below.

Table 5-10 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.

```
call ssyev ( jobz, uplo, n, a, lda, w, work, lwork, info )
call dsyev ( jobz, uplo, n, a, lda, w, work, lwork, info )
```

Discussion

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.

```
call cheev ( jobz, uplo, n, a, lda, w, work, lwork, rwork, info )
call zheev ( jobz, uplo, n, a, lda, w, work, lwork, rwork, info )
```

Discussion

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.

```
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)
```

Discussion

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,

$$Az_i = \lambda_i z_i \quad \text{for } i = 1, 2, \ldots, n.$$

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.

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 $lwork \ge 1$; if *job* = 'N' and $n > 1$, then $lwork \ge 2n+1$; if *job* = 'V' and $n > 1$, then $lwork \ge 3n^2+(5+2k)*n+1$, 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 $liwork \ge 1$; 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 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.

```
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)
```

Discussion

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,

$$Az_i = \lambda_i z_i \quad \text{for } i = 1, 2, \ldots, n.$$

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.

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 $n \le 1$, then $lwork \ge 1$;
if *job* = 'N' and $n > 1$, then $lwork \ge n+1$;
if *job* = 'V' and $n > 1$, then $lwork \ge n^2+2n$

rwork
REAL for cheevd
DOUBLE PRECISION for zheevd
Workspace array, DIMENSION at least *lrwork*.

lrwork
INTEGER. The dimension of the array *rwork*.
Constraints:
if $n \le 1$, 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 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.

```
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)
```

Discussion

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; $vl \le vu$. 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 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*; $ldz \ge 1$. If *jobz* = 'V', then $ldz \ge \max(1,n)$.
lwork	INTEGER. The dimension of the array *work*. Constraint: $lwork \ge \max(1, 8n)$. 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; $0 \le m \le n$. 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.

```
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)
```

Discussion

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* ≤ *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 ≤ *il* ≤ *iu* ≤ *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* ≥ 1. If *jobz* = 'V', then *ldz* ≥ 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 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.

```
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)
```

Discussion

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 ≥ max(1, *n*) if *jobz* = 'V'.

lwork
INTEGER. The dimension of the array *work*.
Constraint: *lwork* ≥ max(1, 26*n*). See *Application notes* for the suggested value of *lwork*.

iwork
INTEGER.
Workspace array, DIMENSION (*liwork*).

liwork
INTEGER. The dimension of the array *iwork*,
lwork ≥ max(1, 10*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, 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.

```
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)
```

Discussion

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: $vl < \lambda_i \leq vu$.
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 ($n \geq 0$).

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: $ldz \ge 1$ if *jobz* = 'N'; $ldz \ge \max(1, n)$ if *jobz* = 'V'.
lwork	INTEGER. The dimension of the array *work*. Constraint: $lwork \ge \max(1, 2n)$. 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* ≥ max(1, 24*n*). .

iwork INTEGER.
Workspace array, DIMENSION (*liwork*).

liwork INTEGER. The dimension of the array *iwork*, *lwork* ≥ max(1, 10*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 ≤ *m* ≤ *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.

```
call sspev (jobz, uplo, n, ap, w, z, ldz, work, info)
call dspev (jobz, uplo, n, ap, w, z, ldz, work, info)
```

Discussion

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* ($n \geq 0$).
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* ≥ 1; if *jobz* = 'V', then *ldz* ≥ 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.

```
call chpev (jobz, uplo, n, ap, w, z, ldz, work, rwork, info)
call zhpev (jobz, uplo, n, ap, w, z, ldz, work, rwork, info)
```

Discussion

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, 2*n*-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 chpev
DOUBLE PRECISION for zhpev.
Workspace array, DIMENSION at least max(1, 3*n*-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.

```
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)
```

Discussion

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,

$$Az_i = \lambda_i z_i \quad \text{for } i = 1, 2, \ldots, n.$$

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.

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* ≥ max(1, *n*) .
lwork	INTEGER. The dimension of the array *work*. Constraints: if *n* ≤ 1, then *lwork* ≥ 1; if *job* = 'N' and *n* > 1, then *lwork* ≥ 2*n*; if *job* = 'V' and *n* > 1, then *lwork* ≥ $2n^2+(5+2k)*n+1$, 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* ≤ 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,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 *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 ?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.

```
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)
```

Discussion

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,

$$Az_i = \lambda_i z_i \quad \text{for } i = 1, 2, \ldots, n.$$

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.

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 — 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* ≥ 1;
if *job* = 'V', then *ldz* ≥ 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* ≥ *n*;
if *job* = 'V' and $n > 1$, then *lwork* ≥ 2*n*

rwork — REAL for chpevd
DOUBLE PRECISION for zhpevd
Workspace array, DIMENSION at least *lrwork*.

lrwork — INTEGER. The dimension of the array *rwork*.
Constraints:
if $n \leq 1$, then *lrwork* ≥ 1;
if *job* = 'N' and $n > 1$, then *lrwork* ≥ *n*;
if *job* = 'V' and $n > 1$, then
lrwork ≥ $3n^2+(4+2k)*n+1$, 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 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.

```
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)
```

Discussion

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* ≥ 1; if *jobz* = 'V', then *ldz* ≥ 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 ε*$||T||_1$ 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.

```
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)
```

Discussion

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* ≥ 1;
if *jobz* = 'V', then *ldz* ≥ 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 ε*$||T||_1$ 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.

```
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)
```

Discussion

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* ≥ 1;
if *jobz* = 'V', then *ldz* ≥ 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.

```
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)
```

Discussion

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.

```
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)
```

Discussion

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 ($n \geq 0$).
kd	INTEGER. The number of super- or sub-diagonals in A ($kd \geq 0$).
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* ≥ 1; if *job* = 'V', then *ldz* ≥ 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* ≥ 2*n*; if *job* = 'V' and $n > 1$, then *lwork* ≥ $3n^2+(4+2k)*n+1$, 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* ≥ 1; if *job* = 'N' and $n > 1$, then *liwork* ≥ 1; if *job* = 'V' and $n > 1$, then *liwork* ≥ 5*n*+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.

```
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)
```

Discussion

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,

$$Az_i = \lambda_i z_i \quad \text{for } i = 1, 2, \ldots, n.$$

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.

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* ≥ 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* ≥ 1; if *job* = 'V', then *ldz* ≥ max(1, *n*) .
lwork	INTEGER. The dimension of the array *work*. Constraints: if *n* ≤ 1, then *lwork* ≥ 1; if *job* = 'N' and *n* > 1, then *lwork* ≥ *n*; if *job* = 'V' and *n* > 1, then *lwork* ≥ $2n^2$
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* ≤ 1, then *lrwork* ≥ 1; if *job* = 'N' and *n* > 1, then *lrwork* ≥ *n*; if *job* = 'V' and *n* > 1, then *lrwork* ≥ $3n^2+(4+2k)*n+1$, 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 *job* = 'N' or $n \leq 1$, then *liwork* ≥ 1;
if *job* = 'V' and $n > 1$, then *liwork* $\geq 5n+2$.

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.

```
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)
```

Discussion

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, $0 \le m \le n$. 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.

```
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)
```

Discussion

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: $1 \le il \le iu \le n$, 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 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: $ldq \ge 1$, $ldz \ge 1$; If *jobz* = 'V', then $ldq \ge \max(1, n)$ and $ldz \ge \max(1, n)$.

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, $0 \le m \le n$. 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.

```
call sstev (jobz, n, d, e, z, ldz, work, info)
call dstev (jobz, n, d, e, z, ldz, work, info)
```

Discussion

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* ≥ 1. If *jobz* = 'V' then *ldz* ≥ 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.

```
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)
```

Discussion

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` ≤ 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 $2^k \geq n$.

`iwork` — `INTEGER`.
Workspace array, `DIMENSION` at least `liwork`.

`liwork` — `INTEGER`. The dimension of the array `iwork`.
Constraints:
if `job` = `'N'` or `n` ≤ 1, then `liwork` ≥ 1;
if `job` = `'V'` and `n` > 1, then `liwork` ≥ 5`n`+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.

```
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)
```

Discussion

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, $0 \le m \le n$. 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.

```
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)
```

Discussion

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: $vl < \lambda_i \leq vu$.
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 ($n \geq 0$).

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: $ldz \ge 1$ if *jobz* = 'N'; $ldz \ge \max(1, n)$ if *jobz* = 'V'.
lwork	INTEGER. The dimension of the array *work*. Constraint: $lwork \ge \max(1, 20n)$.
iwork	INTEGER. Workspace array, DIMENSION (*liwork*).
liwork	INTEGER. The dimension of the array *iwork*, $lwork \ge \max(1, 10n)$.

Output Parameters

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 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 5-12 lists routines described in more detail below.

Table 5-11 Driver Routines for Solving Nonsymmetric Eigenproblems

Routine Name	Operation performed
?gees	Computes the eigenvalues and Shur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Shur form.
?geesx	Computes the eigenvalues and Shur 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 Shur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Shur form.

```
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)
```

Discussion

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 $A = Z T Z^H$.

Optionally, it also orders the eigenvalues on the diagonal of the real-Schur/Shur 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 Shur vectors are not computed.
If *jobvs* = 'V', then Shur 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 Shur 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 ($n \geq 0$).

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 ≥ 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.

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-Shur/Shur 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-Shur 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 Shur 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 Shur 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 ≤ *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 Shur factorization of a general matrix, orders the factorization and computes reciprocal condition numbers.

```
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)
```

Discussion

This routine computes for an *n*-by-*n* real/complex nonsymmetric matrix A, the eigenvalues, the real-Schur/Shur 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/Shur 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 Shur vectors are not computed.
If *jobvs* = 'V', then Shur 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 Shur 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 ($n \geq 0$).
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 $2*sdim*(n-sdim) \leq n*n/2$.

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-Shur/Shur 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-Shur 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 Shur 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 Shur 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 ≤ *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.

```
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)
```

Discussion

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 $\lambda(j)$ 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.

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.

```
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)
```

Discussion

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* ≥ 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*. *For real flavors:* If *sense* = 'N' or 'E', *lwork* ≥ max(1, 2*n*) , and

if *jobvl* = 'V' or *jobvr* = 'V', *lwork* ≥ 3*n*;
If *sense* = 'V' or 'B', *lwork* ≥ *n*(*n*+6).
For good performance, *lwork* must generally be larger.

For complex flavors:
If *sense* = 'N' or 'E', *lwork* ≥ max(1, 2*n*);
If *sense* = 'V' or 'B', *lwork* ≥ n^2+2*n*.
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-Shur/Shur 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 5-12 lists routines described in more detail below.

Table 5-12 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.

```
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)
```

Discussion

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* ≥ 1 ; *ldvt* ≥ 1. If *jobu* = 'S' or 'A', *ldu* ≥ *m* ; If *jobvt* = 'A', *ldvt* ≥ *n*; If *jobvt* = 'S', *ldvt* ≥ min(*m*, *n*).
lwork	INTEGER. The dimension of the array *work*; *lwork* ≥ 1. Constraints: *lwork* ≥ max(3*min(*m*,*n*)+max(*m*,*n*), 5*min(*m*,*n*)) (for real flavors); *lwork* ≥ 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) ≥ *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.

```
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)
```

Discussion

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 ≥ 1 ; *ldvt* ≥ 1.
If *jobz* = 'S' or 'A', or *jobz* = 'O' and $m < n$, then *ldu* ≥ *m* ;
If *jobz* = 'A' or *jobz* = 'O' and $m \geq n$, then *ldvt* ≥ *n*;
If *jobz* = 'S', *ldvt* ≥ min(*m*, *n*).

lwork — INTEGER. The dimension of the array *work*; *lwork* ≥ 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* ≥ *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) ≥ *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* ≥ *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* ≥ *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 ≥ 3**min*(*m*,*n*) + *max* (*max*(*m*,*n*), 6**min*(*m*,*n*));
If jobz = 'O', lwork ≥ 3*(*min*(*m*,*n*))^2 + *max* (*max*(*m*,*n*), 5*(*min*(*m*,*n*))^2 + 4**min*(*m*,*n*));
If jobz = 'S' or 'A', lwork ≥ 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 ≥ 2**min*(*m*,*n*) + *max*(*m*,*n*) ;
If jobz = 'O', lwork ≥ 2*(*min*(*m*,*n*))^2 + *max*(*m*,*n*) + 2**min*(*m*,*n*);
If jobz = 'S' or 'A', lwork ≥ (*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.

```
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)
```

Discussion

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 $m-k-l \geq 0$,

$$D_1 = \begin{array}{c} \\ k \\ l \\ m-k-l \end{array} \begin{array}{c} \begin{array}{cc} k & l \end{array} \\ \begin{pmatrix} I & 0 \\ 0 & C \\ 0 & 0 \end{pmatrix} \end{array}$$

$$D_2 = \begin{array}{c} \\ l \\ p-l \end{array} \begin{array}{c} \begin{array}{cc} k & l \end{array} \\ \begin{pmatrix} 0 & S \\ 0 & 0 \end{pmatrix} \end{array}$$

$$(0 \quad R) = \begin{array}{c} \\ k \\ l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & R_{11} & R_{12} \\ 0 & 0 & R_{22} \end{pmatrix} \end{array}$$

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 $m-k-l < 0$,

$$D_1 = \begin{array}{c} \\ k \\ m-k \end{array} \begin{array}{c} \begin{array}{ccc} k & m-k & k+l-m \end{array} \\ \begin{pmatrix} I & 0 & 0 \\ 0 & C & 0 \end{pmatrix} \end{array}$$

$$D_2 = \begin{array}{c} \\ m-k \\ k+l-m \\ p-l \end{array} \begin{array}{c} \begin{array}{ccc} k & m-k & k+l-m \end{array} \\ \begin{pmatrix} 0 & S & 0 \\ 0 & 0 & I \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$(0 \quad R) = \begin{array}{c} \\ k \\ m-k \\ k+l-m \end{array} \begin{array}{c} \begin{array}{cccc} n-k-l & k & m-k & k+l-m \end{array} \\ \begin{pmatrix} 0 & R_{11} & R_{12} & R_{13} \\ 0 & 0 & R_{22} & R_{23} \\ 0 & 0 & 0 & R_{33} \end{pmatrix} \end{array}$$

where

C = diag (alpha(k+1),...,alpha(m)),
S = diag (beta(k+1),...,beta(m)),
$C^2 + S^2 = I$

On exit, $\begin{pmatrix} R_{11} R_{12} R_{13} \\ 0 \quad R_{22} R_{23} \end{pmatrix}$ 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* ≥ 0).

n — `INTEGER`. The number of columns of the matrices *A* and *B* (*n* ≥ 0).

p — `INTEGER`. The number of rows of the matrix *B* (*p* ≥ 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 5-13 lists routines described in more detail below.

Table 5-13 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.

```
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 )
```

Discussion

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 ($n \geq 0$).

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, $Z^{T}BZ = I$; 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* ≤ *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 ≤ *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*+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.

```
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 )
```

Discussion

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 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 ($n \geq 0$).

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.

```
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 )
```

Discussion

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.

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 $n \leq 1$, $lwork \geq 1$; If *jobz* = 'N' and $n>1$, $lwork \geq 2n+1$; 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

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* ≤ *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 ≤ *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.

?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.

```
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 )
```

Discussion

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 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	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 $n \leq 1$, *lwork* ≥ 1; If *jobz* = 'N' and $n>1$, *lwork* $\geq n+1$; If *jobz* = 'V' and $n>1$, *lwork* $\geq n^2+2n$.
rwork	REAL for chegvd DOUBLE PRECISION for zhegvd. Workspace array, DIMENSION (*lrwork*).
lrwork	INTEGER. The dimension of the array *rwork*. Constraints: If $n \leq 1$, *lrwork* ≥ 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* ≥ 1;
If *jobz* = 'N' and $n>1$, *liwork* ≥ 1;
If *jobz* = 'V' and $n>1$, *liwork* $\geq 5n+3$.

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* $\leq n$, 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 $B = L L^H$.

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.

```
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)
```

Discussion

This routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \quad \text{or} \quad BAx = \lambda x .$$

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* ≥ 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*;
$lwork \ge \max(1, 8n)$.
See *Application Notes* for the suggested value of *lwork*.

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 $info \le n$, 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, $0 \le m \le n$. 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* ≤ *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 ≤ *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

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*$||T||_1$ 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.

```
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)
```

Discussion

This routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \quad \text{or} \quad BAx = \lambda x .$$

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* ($n \ge 0$).

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: $1 \le il \le iu \le n$, 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 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: $ldz \ge 1$; if *jobz* = 'V', $ldz \ge \max(1, n)$.
lwork	INTEGER. The dimension of the array *work*; $lwork \ge \max(1, 2n-1)$. 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 $info \le n$, 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, $0 \le m \le n$. 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* ≤ *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 ≤ *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

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*$||T||_1$ 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.

```
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 )
```

Discussion

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, 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.

```
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 )
```

Discussion

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.

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* ≥ 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* ≤ *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 ≤ *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.

?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.

```
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 )
```

Discussion

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, 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* ≥ 1. If *jobz* = 'V', *ldz* ≥ max(1, *n*) .
lwork	INTEGER. The dimension of the array *work*. Constraints: If $n \leq 1$, *lwork* ≥ 1; If *jobz* = 'N' and *n*>1, *lwork* ≥ 2*n*; If *jobz* = 'V' and *n*>1, *lwork* ≥ $2n^2+6n+1$.
iwork	INTEGER. Workspace array, DIMENSION (*liwork*). .
liwork	INTEGER. The dimension of the array *iwork*. Constraints: If $n \leq 1$, *liwork* ≥ 1; If *jobz* = 'N' and *n*>1, *liwork* ≥ 1; If *jobz* = 'V' and *n*>1, *liwork* ≥ 5*n*+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 $B = U^T U$ 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.

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* $\leq$ *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 $\leq$ *i* $\leq$ *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.

```
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 )
```

Discussion

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* ($n \geq 0$).
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* ≥ 1. If *jobz* = 'V', *ldz* ≥ max(1, *n*) .
lwork	INTEGER. The dimension of the array *work*. Constraints: If $n \leq 1$, *lwork* ≥ 1; If *jobz* = 'N' and *n*>1, *lwork* ≥ *n*; If *jobz* = 'V' and *n*>1, *lwork* ≥ 2*n* .
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* ≥ 1; If *jobz* = 'N' and *n*>1, *lrwork* ≥ *n*; If *jobz* = 'V' and *n*>1, *lrwork* ≥ $2n^2+5n+1$.
iwork	INTEGER. Workspace array, DIMENSION (*liwork*). .

liwork	INTEGER. The dimension of the array *iwork*. Constraints: If $n \leq 1$, *liwork* ≥ 1; If *jobz* = 'N' and $n>1$, *liwork* ≥ 1; If *jobz* = 'V' and $n>1$, *liwork* ≥ $5n+3$.

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* ≤ *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 ≤ *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.

?spgvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage.

```
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 )
```

Discussion

This routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \quad \text{or} \quad BAx = \lambda x .$$

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* ≥ 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 ≤ *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 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* ≥ 1; if *jobz* = 'V', *ldz* ≥ 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* ≤ *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 ≤ *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

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.

```
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 )
```

Discussion

This routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$$Ax = \lambda Bx, \quad ABx = \lambda x, \quad \text{or} \quad BAx = \lambda x .$$

Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite.

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* ≥ 1. If *jobz* = 'V', *ldz* ≥ 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, 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 $B = U^H U$ 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* ≤ *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 ≤ *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

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').

?sbgv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices.

```
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 )
```

Discussion

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 ($n \geq 0$).
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 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 $B = S^T S$, 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 $Z^T B Z = I$.
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 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.

```
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 )
```

Discussion

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 $i \leq n$, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;
if *info* = *n* + *i*, for $1 \leq i \leq n$, 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.

```
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 )
```

Discussion

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* ≥ 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* ≥ 1. If *jobz* = 'V', *ldz* ≥ max(1, *n*) .
lwork	INTEGER. The dimension of the array *work*. Constraints: If *n* ≤ 1, *lwork* ≥ 1; If *jobz* = 'N' and *n*>1, *lwork* ≥ 3*n*; If *jobz* = 'V' and *n*>1, *lwork* ≥ $2n^2+5n+1$.
iwork	INTEGER. Workspace array, DIMENSION (*liwork*). .
liwork	INTEGER. The dimension of the array *iwork*. Constraints: If *n* ≤ 1, *liwork* ≥ 1; If *jobz* = 'N' and *n*>1, *liwork* ≥ 1; If *jobz* = 'V' and *n*>1, *liwork* ≥ 5*n*+3 .

Output Parameters

ab	On exit, the contents of *ab* are overwritten.
bb	On exit, contains the factor *S* from the split Cholesky factorization $B = S^T S$, 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.

```
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 )
```

Discussion

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 ($n \geq 0$).
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* ≥ 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* ≥ 1. If *jobz* = 'V', *ldz* ≥ max(1, *n*) .
lwork	INTEGER. The dimension of the array *work*. Constraints: If *n* ≤ 1, *lwork* ≥ 1; If *jobz* = 'N' and *n*>1, *lwork* ≥ *n*; If *jobz* = 'V' and *n*>1, *lwork* ≥ $2n^2$.
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* ≥ 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* ≥ 1;
If *jobz* = 'N' and $n>1$, *liwork* ≥ 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* ≤ *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.

?sbgvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices.

```
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 )
```

Discussion

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* ≥ 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	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*; $ldq \ge 1$. If *jobz* = 'V', $ldq \ge \max(1, n)$.

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, $0 \le m \le n$. 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* ≤ *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.

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').

?hbgvx

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with banded matrices.

```
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 )
```

Discussion

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* ≥ 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: $1 \le il \le iu \le n$, 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 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*;
$ldz \ge 1$. If *jobz* = 'V', $ldz \ge \max(1, n)$.

ldq — INTEGER. The leading dimension of the output array *q*;
$ldq \ge 1$. If *jobz* = 'V', $ldq \ge \max(1, n)$.

rwork — REAL for chbgvx
DOUBLE PRECISION for zhbgvx.
Workspace array, DIMENSION at least $\max(1, 7n)$.

iwork — INTEGER.
Workspace array, DIMENSION at least $\max(1, 5n)$.

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, $0 \le m \le n$. 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` $\leq$ `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 $\leq$ `i` $\leq$ `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 5-14 lists routines described in more detail below.

Table 5-14 Driver Routines for Solving Generalized Nonsymmetric Eigenproblems

Routine Name	Operation performed
?gges	Computes the generalized eigenvalues, Shur form, and the left and/or right Shur vectors for a pair of nonsymmetric matrices.
?ggesx	Computes the generalized eigenvalues, Shur form, and, optionally, the left and/or right matrices of Shur 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, Shur form, and the left and/or right Shur vectors for a pair of nonsymmetric matrices.

```
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 )
```

Discussion

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 Shur vectors are not computed.
If *jobvsl* = 'V', then the left Shur vectors are computed.

jobvsr CHARACTER*1. Must be 'N' or 'V'.
If *jobvsr* = 'N', then the right Shur vectors are not computed.
If *jobvsr* = 'V', then the right Shur 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 Shur 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* ($n \geq 0$).

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 Shur form *S* .
b	On exit, this array has been overwritten by its generalized Shur 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 Shur 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 Shur 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` ≤ `n` :

the *QZ* iteration failed. (*A*,*B*) is not in Shur 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, Shur form, and, optionally, the left and/or right matrices of Shur vectors .

```
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 )
```

Discussion

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 Shur vectors are not computed.
If *jobvsl* = 'V', then the left Shur vectors are computed.

jobvsr — CHARACTER*1. Must be 'N' or 'V'.
If *jobvsr* = 'N', then the right Shur vectors are not computed.
If *jobvsr* = 'V', then the right Shur 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 Shur 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 Shur form *S* .

b On exit, this array has been overwritten by its generalized Shur 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 Shur 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 Shur 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 ≤ *n* :

the *QZ* iteration failed. (*A*,*B*) is not in Shur 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.

```
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 )
```

Discussion

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 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

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 ≤ *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.

```
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)
```

Discussion

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* = 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

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* ($n \geq 0$).
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* ≥ 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*. *For real flavors:* *lwork* ≥ max(1, 6*n*); if *sense* = 'E', *lwork* ≥ 12*n* ; if *sense* = 'V', or 'B', *lwork* ≥ $2n^2$+ 12*n*+16 . *For complex flavors:*

lwork ≥ max(1, 2*n*);
if *sense* ='N', or 'E', *lwork* ≥ 2*n*;
if *sense* = 'V', or 'B', *lwork* ≥ $2n^2 + 2n$.

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 ≤ *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*).

References

[LUG] E. Anderson, Z. Bai et al. *LAPACK User's Guide.* Third edition, SIAM, Philadelphia, 1999.

[Golub96] G.Golub, C. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, third edition, 1996.

Vector Mathematical Functions

6

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 Math 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 A 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 Release Notes for MKL.

Switching between the two modes (HA and LA) is accomplished by using `vmlSetMode(`*`mode`*`)` (see Table 6-10). 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 6-2, 6-8).

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 6-9).
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	pointer to 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	pointer to 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 A.

Error Diagnostics

The VML library has its own error handler. The only difference for C- and Fortran- interfaces is that the 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 6-12).

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 6-12.
2) Depending on the VML mode, the error handler function invokes:
   - `errno` variable setting. The possible values are shown in the Table 6-1
   - writing error text information to the `stderr` stream
   - raising the appropriate exception on error, if necessary
   - calling the additional error handler callback function.

Table 6-1 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 6-2 lists available mathematical functions and data types associated with them.

Table 6-2 VML Mathematical Functions

Function Short Name	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
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

* continued

Table 6-2 VML Mathematical Functions (continued)

Function Short Name	Data Types	Description
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
Atanh	s, d	Inverse hyperbolic tangent of vector elements

Inv

Performs element by element inversion of the vector.

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`.

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 6-3 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.

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.

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.

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.

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.

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 6-4 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.

Discussion

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).

Exp

Computes an exponential of vector elements.

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 6-5 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.

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.

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.

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.

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.

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
DOUBLE 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.

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.

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.

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.

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 vdAtan
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.

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.

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 6-6 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.

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 6-7 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.

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.

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.

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.

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.

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 A for details on vector indexing methods).

Table 6-8 lists available VML Pack/Unpack functions, together with data types and indexing methods associated with them.

Table 6-8 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.

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.

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 6-9 lists available VML Service functions and their short description.

Table 6-9 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
ClearErrorCallBack	Deletes the additional error handler callback function

SetMode

Sets the VML mode to `mode` *parameter and stores the previous VML mode to* `oldmode`.

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, INTENT(IN). Specifies the former VML mode.

C:

oldmode int. Specifies the former VML mode.

Discussion

The *mode* parameter is designed to control accuracy, FPU and error handling options. Table 6-10 lists values of the *mode* parameter, defined in the mode.h header file for C- interface and in the VML.fi file for Fortran- interface. All other possible values of the *mode* parameter may be obtained from these values by using bitwise OR (|) addition 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 6-10 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
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.

Fortran:

```
mod  = vmlgetmode()
```

C:

```
mod  = vmlGetMode( void );
```

Output Parameters

Fortran:

mod INTEGER. Specifies the full *mode* parameter.

C:

mod int. Specifies the full *mode* parameter.

Discussion

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 6-10. The values of *mode* parameter are defined in the mode.h header file for C- interface and in the VML.fi file for Fortran- interface. You can obtain some of these values using the respective mask from the Table 6-11, 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 6-11 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 VML error status to `err` *and stores the previous VML error status to* `olderr`.

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, INTENT(IN). Specifies the former VML error status.

C:

olderr int. Specifies the former VML error status.

Table 6-12 lists possible values of the *err* parameter.

Table 6-12 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.

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*.

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.

Fortran:

```
oldcallback = vmlseterrorcallback( callback )
```

C:

```
oldcallback = vmlSetErrorCallBack( callback );
```

Input Parameters

Fortran:

callback Pointer to 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 Pointer to the former callback function.

C:

oldcallback Pointer to the former callback function.

Discussion

The callback function is called on each VML mathematical function error if VML_ERRMODE_CALLBACK error mode is set (see Table 6-10).

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.

Fortran:

```
fun = vmlgeterrorcallback( )
```

C:

```
fun = vmlGetErrorCallBack( void );
```

Output Parameters

Fortran:

fun Pointer to the callback function.

C:

fun Pointer to the callback function.

ClearErrorCallBack

Deletes the additional error handler callback function and retrieves the former callback function.

Fortran:

```
oldcallback = vmlclearerrorcallback( )
```

C:

```
oldcallback = vmlClearErrorCallBack( void );
```

Output Parameters

Fortran:

oldcallback INTEGER. Pointer to the former callback function.

C:

oldcallback int. Pointer to the former callback function.

Vector Generators of Statistical Distributions

7

This chapter describes the part of MKL which is known as vector statistics library (VSL) and is designed for the purpose of generating vectors of pseudorandom numbers.

VSL provides a set of pseudorandom number generator subroutines implementing basic continuous and discrete distributions. 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 6).

All VSL subroutines can be classified into three major categories:

- Pseudorandom number generators for different types of statistical distributions, for example, uniform, normal (Gaussian), binomial, etc. Detailed description of the generators can be found in Pseudorandom Generators section.
- Basic 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 discussion, a Random Number Generator (RNG) means a number-theoretic deterministic algorithm that generates number sequences, which can be interpreted as random samplings from a universal set with a given probability distribution function. Since random numbers are generated by a deterministic algorithm, they cannot be truly random and should be referred to as pseudorandom. The respective generators should be also called pseudorandom. However, in this chapter no specific differentiation is made between random and pseudorandom numbers, as well as between random and pseudorandom generators unless the context requires otherwise. Likewise, the terms *random number* and *variate*, *statistical distribution* and *probability distribution*, are not distinguished here either.

The choice of a number-theoretic algorithm *A* and initial conditions *I* identifies a unique sequence of random numbers, which is called a random stream. The pair $\langle A, I \rangle$ is referred to as the random stream state. In VSL a stream is identified by a *stream descriptor* represented as `TYPE (VSL_STREAM_STATE)` structure in FORTRAN interface, and `VSLStreamStatePtr` pointer in C interface.

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. The most common transformation techniques include the inverse Cumulative Distribution Function (CDF), acceptance/rejection method, and mixtures. For certain types of distribution, several generation methods are implemented.

VSL subroutines for pseudorandom number generation accept the stream descriptor and the distribution parameters as input and write the result in a vector of pseudorandom numbers with a given distribution. For a given statistical distribution, several generation methods can be used, which may differ in efficiency for particular ranges of input parameters. Consequently, the most efficient generators often use different methods for different ranges. To establish the generation method to be used in the subroutine, you

should specify the input parameter called the method number. Description of methods available for each generator can be found in Pseudorandom Generators section.

In the discussion that follow, the terms *multiprocessor system*, *computational node*, and *processor* refer to any configuration of the system with shared or distributed memory, or combination of the two. Specifically, a computational node, or a processor, refers to a computational unit capable of performing independent parallel computations (this may be either a physical processor, a cluster node, or a logical parallel process).

Mathematical Notation

The following notation is used throughout the text:

N	The set of natural numbers $N = \{1, 2, 3\ldots\}$.
Z	The set of integers $Z = \{\ldots -3, -2, -1, 0, 1, 2, 3 \ldots\}$.
R	The set of real numbers.
$\lfloor a \rfloor$	The floor of a (the largest integer less than or equal to a).
$\oplus$ 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 $x_{n+1} = (ax_n + c)\ mod\ m$, 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 pseudorandom 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 implementing a given type of random number 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 Pseudorandom 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 pseudorandom 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* pseudorandom numbers.

Additional parameters included into *<distribution parameters>* field are individual for each generator subroutine and are described in detail in Pseudorandom Generators section.

To invoke a pseudorandom generator, use a call to the respective VSL subroutine. For example, to obtain a vector *r*, composed of *n* independent and identically distributed pseudorandom 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 Pseudorandom Generators

Basic Random Number Generators (BRNGs) are the major and widely spread tool to obtain uniformly distributed pseudorandom numbers.

VSL provides a number of basic generators that differ in speed and quality: the 32-bit multiplicative congruential generator *MCG(1132489760, 2^{31} - 1)* [L'Ecuyer99], the 32-bit generalized feedback shift register generator *GFSR(250,103)* [Kirkpatrick81], and the combined multiple recursive generator *MRG-32k3a* [L'Ecuyer99a]. Essentially, applicability of a basic generator to a given computational task is very difficult to estimate. To ensure more reliable results, basic generators are usually tested in a series of statistical tests prior to actual computation. Comparative performance analysis of the generators and testing results can be found in VSLNotes.

Users may want to design and use their own basic generators. 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. 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 VSLNotes.

Random Streams

Several random streams may be used in one application for a number of reasons.

First, it may be necessary to supply random data to different computational nodes of a multiprocessor system. In this case, the following options are available:

- use an individual basic generator for each computational node, so that each random stream is filled from a different basic generator;
- use one basic generator for all computational nodes and generate several independent random streams using the leapfrog method or the block-splitting method;

- use combination of the two approaches, when one basic generator is used to generate independent streams for all nodes and each of the nodes in turn uses its own generator.

Another reason is related to the fact that many Monte Carlo simulations require additional randomization. A simple illustration is the necessity to assign random streams to different elements of the model or to run variance reduction methods [Bratley87].

In either case, the correlation between different random streams can affect reliability of the final result.

Data Types

Stream State. Random numbers can be generated by portions using the notion of a *stream state*, which is a structure created after a call to the stream creating subroutine. A stream state descriptor is used to access the structure:

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 7-1 lists all available service subroutines

Table 7-1 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.
DeleteStream	Deletes previously created stream.
CopyStream	Copies a stream to another stream.
LeapfrogStream	Initializes the stream of *k*-th computational node in a *nstreams*-node cluster by the leapfrog method.
SkipAheadStream	Initializes the stream by the block-splitting method.
GetStreamStateBrng	Obtains the index of the basic generator responsible for the generation of a given random stream.

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:

4. Creating and initializing a stream (NewStream, NewStreamEx, CopyStream, LeapfrogStream, SkipAheadStream).
5. Generating pseudorandom numbers with given distribution, see Pseudorandom Generators.
6. Deleting the stream (DeleteStream).

Note that you can concurrently create multiple streams and obtain pseudorandom 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.

Fortran:

```
call vslnewstream( stream, brng, seed )
```

C:

```
vslNewStream( stream, brng, seed )
```

Discussion

For a basic generator with number *brng*, this function creates a new stream and initializes it with a 32-bit seed. The function is also applicable for generators with multiple initial conditions. See VSLNotes 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.

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.

Fortran:

```
call vslnewstreamex( stream, brng, n, params )
```

C:

```
vslNewStreamEx( stream, brng, n, params  )
```

Discussion

This function provides an advanced tool to set the initial conditions for a basic generator if its input arguments imply several initialization parameters. This subroutine should not be used unless it is specially necessary. Whenever possible, use vslNewStream, which is analogous to vslNewStreamEx except that it takes only one 32-bit initial condition. In particular, vslNewStreamEx may be used to initialize the seed tables in Generalized Feedback Shift Register Generators (GFSRs). A more detailed description of this issue can be found in VSLNotes.

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.

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.

Fortran:

```
call vsldeletestream( stream )
```

C:

```
vslDeleteStream( stream )
```

Discussion

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.

Fortran:

```
call vslcopystream( newstream, srcstream )
```

C:

```
vslCopyStream( newstream, srcstream )
```

Discussion

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.

LeapfrogStream

Initializes stream of k-th computational node in nstreams-node cluster using the leapfrog method.

Fortran:

```
call vslleapfrogstream( stream, k, nstreams )
```

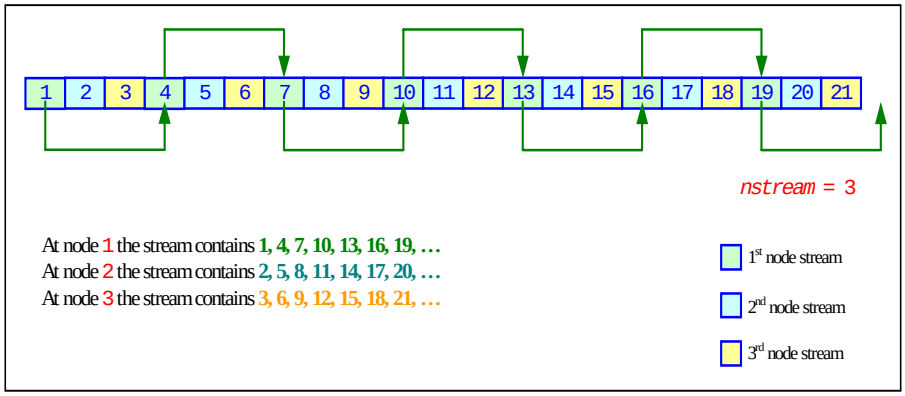
C:

```
vslLeapfrogtream( stream, k, nstreams )
```

Discussion

The function uses the leapfrog method (see Figure 7-1) to generate an independent random stream for the computational node k, $0 \le k < nstreams$, where *nstreams* is the largest number of computational nodes used.

Figure 7-1 Leapfrog Method



The following code examples illustrate the initialization of three independent streams using the leapfrog method:

Example 7-1 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)

! 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 7-2 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 number of independent streams.

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 number of independent steams.

SkipAheadStream

Initializes a stream using the block-splitting method.

Fortran:

```
call vslskipaheadstream( stream, nskip )
```

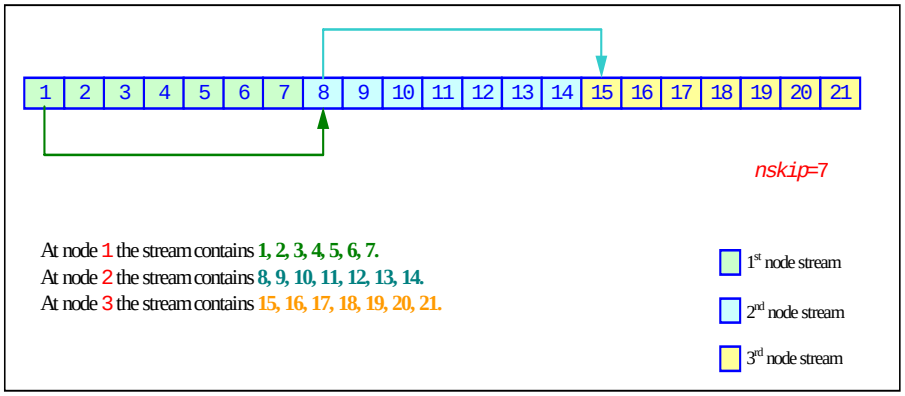
C:

```
vslSkipAheadStream( stream, nskip )
```

Discussion

This function initializes an independent random stream of a given computational node through the block-splitting method (see Figure 7-2). The maximum number of computational nodes is unlimited. The largest number of elements skipped in a given stream is *nskip*.

Figure 7-2 Block-Splitting Method



The following code examples illustrate how to initialize three independent streams using the `SkipAheadStream` function:

Example 7-3 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);
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 7-4 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.

Fortran:

```
brng = vslgetstreamstatebrng( stream )
```

C:

```
brng = vslGetStreamStateBrng( stream )
```

Discussion

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.

Pseudorandom 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 7-2 and Table 7-3 list the pseudorandom number generator subroutines, together with used data types and output distributions.

Table 7-2 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.
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 7-3 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.
Binomial	i	Binomial distribution.
Hypergeometric	i	Hypergeometric distribution.
Poisson	i	Poisson distribution.
NegBinomial	i	Negative binomial distribution, or Pascal distribution.

Continuous Distributions

This section describes routines for generating pseudorandom numbers with continuous distribution.

Uniform

Generates pseudorandom numbers with uniform distribution.

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 )
```

Discussion

This function generates pseudorandom 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:

$$f_{a,b}(x) = \begin{cases} \frac{1}{b-a}, & x \in (a, b) \\ 0, & x \notin (a, b) \end{cases}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,b}(x) = \begin{cases} 0, & x < a \\ \frac{x-a}{b-a}, & a \le x < b \\ 1, & x \ge b \end{cases}, -\infty < x < +\infty.$$

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* pseudorandom numbers uniformly distributed over the interval `(a,b)`.

C:

r	`float*` for `vsRngUniform`. `double*` for `vdRngUniform`. Vector of *n* pseudorandom numbers uniformly distributed over the interval `(a,b)`.

Gaussian

Generates normally distributed pseudorandom numbers.

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 )
```

Discussion

This function generates pseudorandom 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 pseudorandom numbers.

C:

r	float* for vsRngGaussian. double* for vdRngGaussian. Vector of *n* normally distributed pseudorandom numbers.

Exponential

Generates exponentially distributed pseudorandom numbers.

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 )
```

Discussion

This function generates pseudorandom 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:

$$f_{a,\beta}(x) = \begin{cases} \frac{1}{\beta}\exp((-(x-a))/\beta), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp((-(x-a))/\beta), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

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 pseudorandom numbers.

C:

r	`float*` for `vsRngExponential`. `double*` for `vdRngExponential`. Vector of *n* exponentially distributed pseudorandom numbers.

Laplace

Generates pseudorandom numbers with Laplace distribution.

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 )
```

Discussion

This function generates pseudorandom 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 pseudorandom numbers.

C:

r	`float*` for `vsRngLaplace`. `double*` for `vdRngLaplace`. Vector of *n* Laplace distributed pseudorandom numbers.

Weibull

Generates Weibull distributed pseudorandom numbers.

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 )
```

Discussion

This function generates Weibull distributed pseudorandom 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:

$$f_{a,\alpha,\beta}(x) = \begin{cases} \frac{\alpha}{\beta^{\alpha}}(x-a)^{\alpha-1}\exp\left(-\left(\frac{x-a}{\beta}\right)^{\alpha}\right), & x \geq a \\ 0, & x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\alpha,\beta}(x) = \begin{cases} 1-\exp\left(-\left(\frac{x-a}{\beta}\right)^{\alpha}\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

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 pseudorandom numbers.

C:

r	`float*` for `vsRngWeibull`. `double*` for `vdRngWeibull`. Vector of *n* Weibull distributed pseudorandom numbers.

Cauchy

Generates Cauchy distributed pseudorandom values.

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 )
```

Discussion

This function generates Cauchy distributed pseudorandom 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 pseudorandom numbers.

C:

r — `float*` for `vsRngCauchy`.
`double*` for `vdRngCauchy`.
Vector of *n* Cauchy distributed pseudorandom numbers.

Rayleigh

Generates Rayleigh distributed pseudorandom values.

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 )
```

Discussion

This function generates Rayleigh distributed pseudorandom 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:

$$f_{a,\beta}(x) = \begin{cases} \frac{2(x-a)}{\beta^2}\exp\left(-\frac{(x-a)^2}{\beta^2}\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp\left(-\frac{(x-a)^2}{\beta^2}\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

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 pseudorandom numbers.

C:

r	`float*` for `vsRngRayleigh`. `double*` for `vdRngRayleigh`. Vector of *n* Rayleigh distributed pseudorandom numbers.

Lognormal

Generates lognormally distributed pseudorandom numbers.

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 pseudorandom 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 pseudorandom numbers.

C:

r	`float*` for `vsRngLognormal`. `double*` for `vdRngLognormal`. Vector of *n* lognormally distributed pseudorandom numbers.

Gumbel

Generates Gumbel distributed pseudorandom values.

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 )
```

Discussion

This function generates Gumbel distributed pseudorandom 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* pseudorandom values with Gumbel distribution.

C:

r	float* for vsRngGumbel. double* for vdRngGumbel. Vector of *n* pseudorandom values with Gumbel distribution.

Discrete Distributions

This section describes routines for generating pseudorandom numbers with discrete distribution.

Uniform

Generates pseudorandom numbers uniformly distributed over the interval $[a, b)$.

Fortran:

```
call virnguniform( method, stream, n, r, a, b )
```

C:

```
viRngUniform( method, stream, n, r, a, b )
```

Discussion

This function generates pseudorandom 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:

$$F_{a,b}(x) = \begin{cases} 0, & x < a \\ \frac{\lfloor x - a + 1 \rfloor}{b - a}, & a \le x < b, x \in R. \\ 1, & x \ge b \end{cases}$$

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* pseudorandom values uniformly distributed over the interval [*a*,*b*).

C:

r	int*. Vector of *n* pseudorandom values uniformly distributed over the interval [*a*,*b*).

UniformBits

Generates integer random values with uniform bit distribution.

Fortran:

```
call virnguniformbits( method, stream, n, r )
```

C:

```
viRngUniform( method, stream, n, r )
```

Discussion

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 truly random. See VSLNotes 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* pseudorandom 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` pseudorandom 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 pseudorandom values.

Fortran:

```
call virngbernoulli( method, stream, n, r, p )
```

C:

```
viRngBernoulli( method, stream, n, r, p )
```

Discussion

This function generates Bernoulli distributed pseudorandom 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 pseudorandom values.

C:

r	`int*`. Vector of *n* Bernoulli distributed pseudorandom values.

Geometric

Generates geometrically distributed pseudorandom values.

Fortran:

```
call virnggeometric( method, stream, n, r, p )
```

C:

```
viRngGeometric( method, stream, n, r, p )
```

Discussion

This function generates geometrically distributed pseudorandom 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:

$$F_p(x) = \begin{cases} 0, & x < 0 \\ 1 - (1 - p)^{\lfloor x + 1 \rfloor}, & x \geq 0 \end{cases}, x \in R.$$

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 pseudorandom values.

C:

r	int*. Vector of *n* geometrically distributed pseudorandom values.

Binomial

Generates binomially distributed pseudorandom numbers.

Fortran:

```
call virngbinomial( method, stream, n, r, ntrial, p )
```

C:

```
viRngBinomial( method, stream, n, r, ntrial, p )
```

Discussion

This function generates binomially distributed pseudorandom 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:

$$P(X = k) = C_m^k p^k (1-p)^{m-k}, \; k \in \{0, 1, \ldots, m\}.$$

The cumulative distribution function is as follows:

$$F_{m,p}(x) = \begin{cases} 0, & x < 0 \\ \sum_{k=0}^{\lfloor x \rfloor} C_m^k p^k (1-p)^{m-k}, & 0 \le x < m \\ 1, & x \ge m \end{cases}, \; x \in R.$$

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 pseudorandom values.

C:

r	int*. Vector of *n* binomially distributed pseudorandom values.

Hypergeometric

Generates hypergeometrically distributed pseudorandom values.

Fortran:

```
call virnghypergeometric( method, stream, n, r, l, s, m )
```

C:

```
viRngHypergeometric( method, stream, n, r, l, s, m )
```

Discussion

This function generates hypergeometrically distributed pseudorandom 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:

$$F_{l,s,m}(x) = \begin{cases} 0, & x < max(0, s + m - l) \\ \sum_{k=max(0, s+m-l)}^{\lfloor x \rfloor} \frac{C_m^k C_{l-m}^{s-k}}{C_l^s}, & max(0, s + m - l) \leq x \leq min(s, m) \\ 1, & x > min(s, m) \end{cases}$$

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 pseudorandom values.

C:

r	int*. Vector of *n* hypergeometrically distributed pseudorandom values.

Poisson

Generates Poisson distributed pseudorandom values.

Fortran:

```
call virngpoisson( method, stream, n, r, lambda )
```

C:

```
viRngPoisson( method, stream, n, r, lambda )
```

Discussion

This function generates Poisson distributed pseudorandom 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:

$$F_\lambda(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda^k e^{-\lambda}}{k!}, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in R.$$

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). 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 pseudorandom values.

C:

r	int*. Vector of *n* Poisson distributed values.

NegBinomial

Generates pseudorandom numbers with negative binomial distribution.

Fortran:

```
call virngnegbinomial( method, stream, n, r, a, p )
```

C:

```
viRngNegBinomial( method, stream, n, r, a, p )
```

Discussion

This function generates pseudorandom 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:

$$P(X = k) = C^{k}_{a+k-1} p^{a} (1-p)^{k}, k \in \{0, 1, 2, \ldots\}.$$

The cumulative distribution function is as follows:

$$F_{a,p}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} C^{k}_{a+k-1} p^{a} (1-p)^{k}, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in R.$$

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* pseudorandom values with negative binomial distribution.

C:

r	int*. Vector of *n* pseudorandom values with negative binomial distribution.

Advanced Service Subroutines

This section describes service subroutines for registering a basic generator and obtaining properties of the previously registered basic generators.

Table 7-4 Advanced Service Subroutines

Subroutine	Short Description
RegisterBrng	Registers a user-designed basic generator.
GetBrngProperties	Returns the structure with properties of the basic generator with a given number.

Data types

The subroutines of this section refer to a structure defining the properties of the basic generator:

Example 7-5 Fortran Version

```
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
```

Example 7-6 C Version

```
typedef struct _VSLBRngProperties {
            int             StreamStateSize;
            int             NSeeds;
            int             IncludesZero;
            int             WordSize;
            int             NBits;
            InitStreamPtr   InitStream;
            sBRngPtr        sBRng;
            dBRngPtr        dBRng;
            iBRngPtr        iBRng;
} VSLBRngProperties;
```

Example 7-7 Pointers to Functions

```
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[] );
```

Table 7-5 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 pseudorandom 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.

FORTRAN:
`nbits`
C:
`NBits`
	The number of bits required to represent a pseudorandom value in integer arithmetic. Note that, for instance, 48-bit pseudorandom 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 pseudorandom 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).

Table 7-5 Field Descriptions (continued)

Field	Short Description

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 pseudorandom 0. On the other hand, generators like multiplicative congruential generators never generate such a number. In most cases this information is irrelevant because the probability of generating a zero value is extremely small. However, in certain non-uniform distribution generators the possibility for a basic generator to produce a pseudorandom 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 pseudorandom 0.
2. A specific generator that permits operations over single bits and bit groups of pseudorandom numbers.

RegisterBrng

Registers user-defined basic generator.

Fortran:

```
brng = vslregisterbrng( properties )
```

C:

```
brng = vslRegisterBrng( properties )
```

Discussion

An example of a registration procedure can be found in 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.

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 VSL\Examples.

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 */
```

Discussion

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     reserved1
       INTEGER     reserved2
       [ fields specific for the given generator ]
end type mystream_state
```

C:

```
typedef struct
{
       int     Reserved1;
       int     Reserved2;
       [ 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 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 pseudorandom 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 */
```

References

[Bratley87] Bratley P., Fox B.L., and Schrage L.E. *A Guide to Simulation*. 2nd edition. Springer-Verlag, New York, 1987.

[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.

Advanced DFT Interface 8

The Fast Fourier Transform (FFT) algorithm that calculates the Discrete Fourier Transform (DFT) is one of the major breakthroughs in scientific computing and is now an indispensable tool in a vast number of fields. Unfortunately, software that provide fast computation of DFT via FFT differ vastly in functionality and lack uniformity. A widely accepted Applications Programmer Interface (API) for DFT would advance the field of scientific computing significantly. In this chapter, we present the specification of DFTI, a new interface that combines functionality with ease of use.

Although MKL still supports the FFT interface described in chapter 3, users are encouraged to migrate to the new advanced DFTI interface in their application programs.

Introduction

In this chapter, we present the Fortran and C specification of the DFTI interface. Fortran stands for Fortran 95.

NOTE. *DFTI interface relies critically on many modern features offered in Fortran 95 that have no counterpart in Fortran 77.*

We assume the availability of native complex types in C as they are specified in C9X. Before presenting the details of the specification, we give a couple of usage examples.

For most common situations, we expect a DFT computation can be effected by three function calls. Here are two one-dimensional computations.

```
// Fortran example.
// 1D complex to complex, and real to conjugate even
Use 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_Ptr, X )
// 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_Ptr, Y)
// 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).

/* C example, float _Complex is defined in C9X */
#include "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);
/* 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);
/* 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.

```
// Fortran example.
// 2D complex to complex, and real to conjugate even
Use DFTI
Complex :: X(32,100)
Real :: Y(34,100)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc1_Handle, My_Desc2_Handle
Integer :: Status, L(2)
...put input data into X(j,k), Y(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)
// result is given by X(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)
// result is given by the complex value z(j,k) 1<=j<=32; 1<=k<=100 where
// z(j,k) = Y(2j-1,k) + iY(2j,k) 1<=j<=17; 1<=k<=100
// z(j,k) = Y(2(34-j)-1,k) - iY(2(34-j),k) 18<=j<=32; 1<=k<=100
```

```
/* C example */
#include "dfti.h"
float _Complex x[32][100];
float y[34][100];
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
...put 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[0]);
/* 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[0]);
/* 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 record of type `DFTI_DESCRIPTOR`, when created, contains information about the length and domain of the DFT to be computed. Moreover, it contains the setting of a rather large number of configuration parameters. The illustrations above use the default settings for all of these parameters, which 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. For example, suppose you would prefer to preserve the input data after the DFT computation. To do that, you should change the configuration of the placement of result to that of "not in place" from the default choice of "in place."

```
// Fortran example
// 1D complex to complex, not in place
Use 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)
// result is X_out(1),X_out(2),...,X_out(32)

/* C example */
#include "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);
/* result is x_out[0], x_out[1], ..., x_out[31] */
```

The approach uses one single data structure, the descriptor, to record flexible configuration whose parameters can be changed independently. This results in vast functionality and ease of use.

DFTI

The interface is called DFTI. In Fortran, it is provided by the module `DFTI` and accessed through the "`use`" statement in Fortran. In C, it is provided by the header file `dfti.h` and accessed through "`include`". This interface provides a number of types, parameters, and functions. 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.

There are four main categories of functions.

1. **Descriptor Manipulation.** There are four functions in this category. The first one 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 "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 makes an extra copy of a descriptor, and the fourth function frees up all the memory allocated for the descriptor information.
2. **DFT Computation**. There are two functions in this category. The first effects a forward DFT computation, and the second a backward DFT computation.
3. **Descriptor configuration**. There are two functions in this category. One function sets one specific value to one of the many configuration parameters that are changeable (a few are not); the other 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**. All of the functions 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 a latter stage, DFTI at present provides for one logical status class function and a simple status message generation function.

In consideration for mixed-language programming, the specification mandates that the function in Fortran is provided by an interface body, thus separating the function name from the external names. This is mandated even for the functions that are not generic in nature. However, the actual external names and details of the interface body is not part of the specification, although some examples are given in this document. We now specify each of these functions in the order of the categories.

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.

Usage

```
// Fortran
Status = DftiCreateDescriptor( Desc_Handle,      &
                               Precision,        &
                               Forward_Domain,   &
                               Dimension,        &
                               Length )

/* C */
```

```
status = DftiCreateDescriptor( *desc_handle,
                               precision,
                               forward_domain,
                               dimension,
                               length );
```

Discussion

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 such as FFTPACK or NAG. 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 DFTI 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 8-3 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. A zero status value indicates a successful completion. See “Status Checking” for more information on returned status.

Interface and prototype

```
//Fortran interface.
INTERFACE DftiCreateDescriptor
//Note that the body provided here is for illustration only
//The specification does not mandate what it should be.
//It is possible that implementation will use common external
//functions below to be shared by Fortran and C implementation of DFTI
```

```
  FUNCTION DFTI_CREATE_DESCRIPTOR_1D( Desc_Handle, Prec, Dom, Dim, Length )
     INTEGER :: DFTI_CREATE_DESCRIPTOR_1D
     TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
     INTEGER, INTENT(IN) :: Prec, Dom
     INTEGER, INTENT(IN) :: Dim, Length
  END FUNCTION DFTI_CREATE_DESCRIPTOR_1D

  FUNCTION DFTI_CREATE_DESCRIPTOR_HIGHD( Desc_Handle, Prec, Dom, Dim, Length )
     INTEGER :: DFTI_CREATE_DESCRIPTOR_HIGHD
     TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
     INTEGER, INTENT(IN) :: Prec, Dom
     INTEGER, INTENT(IN) :: Dim, Length(*)
  END FUNCTION DFTI_CREATE_DESCRIPTOR_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.

Usage

```
// Fortran
Status = DftiCommitDescriptor( Desc_Handle )
```

```
/* C */
status = DftiCommitDescriptor( desc_handle );
```

Discussion

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. In all likelihood, this committal function call will be immediately followed by a computation function call (see "DFT Computation").

A zero status value indicates a successful completion. See "Status Checking" for more information on returned status.

Interface and prototype

```
// Fortran interface
INTERFACE DftiCommitDescriptor
//Note that the body provided here is for illustration only
//The specification does not mandate what it should be.
//It is possible that implementation will use common external
//functions below to be shared by Fortran and C implementation of DFTI
 FUNCTION DFTI_COMMIT_DESCRIPTOR_EXTERNAL( Desc_Handle )
    INTEGER :: DFTI_COMMIT_DESCRIPTOR_EXTERNAL
    TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 END FUNCTION DFTI_COMMIT_DESCRIPTOR_EXTERNAL
END INTERFACE DftiCommitDescriptor

/* C prototype */
long DftiCommitDescriptor( DFTI_DESCRIPTOR_HANDLE );
```

CopyDescriptor

Copies an existing descriptor.

Usage

```
// Fortran
Status = DftiCopyDescriptor( Desc_Handle_Original,
                             Desc_Handle_Copy )
/* C */
status = DftiCopyDescriptor( desc_handle_original,
                             &desc_handle_copy );
```

Discussion

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 to be specified next.
A zero status value indicates a successful completion. See "Status Checking" for more information on returned status.

Interface and prototype

```
// Fortran interface
INTERFACE DftiCopyDescriptor
//Note that the body provided here is for illustration only
//The specification does not mandate what it should be.
//It is possible that implementation will use common external
//functions below to be shared by Fortran and C implementation of DFTI
 FUNCTION DFTI_COPY_DESCRIPTOR_EXTERNAL( Desc_Handle_Original,
                                         Desc_Handle_Copy )
   INTEGER :: DFTI_COPY_DESCRIPTOR_EXTERNAL
   TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle_Original, Desc_Handle_Copy
 END FUNCTION DFTI_COPY_DESCRIPTOR_EXTERNAL
END INTERFACE DftiCopyDescriptor

/* C prototype */
long DftiCopyDescriptor( DFTI_DESCRIPTOR_HANLDE, DFTI_DESCRIPTOR_HANDLE * );
```

FreeDescriptor

Frees memory allocated for a descriptor.

Usage

```
// Fortran
Status = DftiFreeDescriptor( Desc_Handle )
/* C */
status = DftiFreeDescriptor( &desc_handle );
```

Discussion

This function frees up all memory space allocated for a descriptor. A zero status value indicates a successful completion. See "Status Checking" for more information on returned status.

Interface and prototype

```
// Fortran interface
INTERFACE DftiFreeDescriptor
//Note that the body provided here is for illustration only
//The specification does not mandate what it should be.
//It is possible that implementation will use common external
//functions below to be shared by Fortran and C implementation of DFTI
 FUNCTION DFTI_FREE_DESCRIPTOR_EXTERNAL( Desc_Handle )
    INTEGER :: DFTI_FREE_DESCRIPTOR_EXTERNAL
    TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 END FUNCTION DFTI_FREE_DESCRIPTOR_EXTERNAL
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.

Usage

```
// 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 );
```

Discussion

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.

A zero status value indicates a successful completion. See "Status Checking" for more information on returned status.

Interface and prototype

```
//Fortran interface.
INTERFACE DftiComputeFoward
//Note that the body provided here is for illustration only
//The specification does not mandate what it should be.
//It is possible that implementation will use common external
//functions below to be shared by Fortran and C implementation of DFTI
 // One argument single precision complex
 FUNCTION DFTI_COMPUTE_FORWARD_C( Desc_Handle, X )
    INTEGER :: DFTI_COMPUTE_FORWARD_C
    TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    COMPLEX, INTENT(INOUT) :: X(*)
 END FUNCTION DFTI_COMPUTE_FORWARD_C
 // One argument double precision complex
 FUNCTION DFTI_COMPUTE_FORWARD_Z( Desc_Handle, X )
    INTEGER :: DFTI_COMPUTE_FORWARD_Z
    TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    COMPLEX (Kind((0D0,0D0))), INTENT(INOUT) :: X(*)
 END FUNCTION DFTI_COMPUTE_FORWARD_Z
 // One argument single precision real
 FUNCTION DFTI_COMPUTE_FORWARD_R( Desc_Handle, X )
    INTEGER :: DFTI_COMPUTE_FORWARD_R
    TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    REAL, INTENT(INOUT) :: X(*)
 END FUNCTION DFTI_COMPUTE_FORWARD_R
 // One argument double precision real
 ...
 // Two argument single precision complex
 ...
 ...
 // Four argument double precision real
 FUNCTION DFTI_COMPUTE_FORWARD_DDDD( Desc_Handle, X1_In, X2_In,
                                     Y1_Out, Y2_Out )
    INTEGER :: DFTI_COMPUTE_FORWARD_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 DFTI_COMPUTE_FORWARD_DDDD
END INTERFACE DftiComputeFoward

/* C prototype */
long DftiComputeForward( DFTI_DESCRIPTOR_HANDLE,
                         void *,
                         ... );
```

The implementations of DFTI 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.

Usage

```
// 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 );
```

Discussion

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.

A zero status value indicates a successful completion. See "Status Checking" for more information on returned status.

Interface and prototype

```
//Fortran interface.
INTERFACE DftiComputeBackward
//Note that the body provided here is for illustration only
//The specification does not mandate what it should be.
//It is possible that implementation will use common external
//functions below to be shared by Fortran and C implementation of DFTI
 // One argument single precision complex
 FUNCTION DFTI_COMPUTE_BACKWARD_C( Desc_Handle, X )
    INTEGER :: DFTI_COMPUTE_BACKWARD_C
    TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    COMPLEX, INTENT(INOUT) :: X(*)
 END FUNCTION DFTI_COMPUTE_BACKWARD_C
 // One argument double precision complex
 FUNCTION DFTI_COMPUTE_BACKWARD_Z( Desc_Handle, X )
    INTEGER :: DFTI_COMPUTE_BACKWARD_Z
    TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    COMPLEX (Kind((0D0,0D0))), INTENT(INOUT) :: X(*)
 END FUNCTION DFTI_COMPUTE_BACKWARD_Z
 // One argument single precision real
 FUNCTION DFTI_COMPUTE_BACKWARD_R( Desc_Handle, X )
    INTEGER :: DFTI_COMPUTE_BACKWARD_R
    TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    REAL, INTENT(INOUT) :: X(*)
```

```
 END FUNCTION DFTI_COMPUTE_BACKWARD_R
 // One argument double precision real
 ...
 // Two argument single precision complex
 ...
 ...
 // Four argument double precision real
 FUNCTION DFTI_COMPUTE_BACKWARD_DDDD( Desc_Handle, X1_In, X2_In,
                                      Y1_Out, Y2_Out )
   INTEGER :: DFTI_COMPUTE_BACKWARD_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 DFTI_COMPUTE_BACKWARD_DDDD
END INTERFACE DftiComputeBackward


/* C prototype */
long DftiComputeBackward( DFTI_DESCRIPTOR_HANDLE,
                          void *,
                          ... );
```

The implementations of DFTI 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 sets one particular configuration parameter to an appropriate value, and the value getting function 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.

Table 8-1 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

Table 8-1 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

Every single one of these configuration parameters is identified by a named constant in the 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 8-1; the list of configuration parameters that are read-only is given in Table 8-2. Note that all parameters are readable. Most of these parameters are self-explanatory, while some others are discussed more fully with the specification of the relevant functions.

Table 8-2 Read-Only Configuration Parameters

Named Constants	Value Type	Comments
DFTI_COMMIT_STATUS	Name constant	Whether descriptor has been committed
DFTI_VERSION	String	DFTI implementation 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 8-3.

Table 8-3 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)

Table 8-3 Named Constant Configuration Values (continued)

Named Constant	Comments
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_VERSION_LENGTH	Number of characters for version length
DFTI_MAX_NAME_LENGTH	Maximum descriptor name length
DFTI_MAX_MESSAGE_LENGTH	Maximum status message length

Table 8-4 lists the possible values for those configuration parameters that are discrete in nature.

Table 8-4 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)

Table 8-5 lists the default values of the settable configuration parameters.

Table 8-5 Default Configuration Values of Settable Parameters

Named Constants	Default Value
DFTI_NUMBER_OF_TRANSFORMS	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_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

SetValue

Sets one particular configuration parameter with the specified configuration value.

Usage

```
// Fortran
Status = DftiSetValue( Desc_Handle,        &
                       Config_Param,       &
                       Config_Val )
/* C */
status = DftiSetValue( desc_handle,
                       config_param,
                       config_val );
```

Discussion

This function sets one particular configuration parameter with the specified configuration value. The configuration parameter is one of the named constants listed in Table 8-1, 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.

A zero status value indicates a successful completion. See "Status Checking" for more information on returned status.

Interface and prototype

```
// Fortran interface
INTERFACE DftiSetValue
//Note that the body provided here is for illustration only
//The specification does not mandate what it should be.
//It is possible that implementation will use common external
//functions below to be shared by Fortran and C implementation of DFTI
```

```
  FUNCTION DFTI_SET_VALUE_INTVAL( Desc_Handle, Config_Param, INTVAL )
     INTEGER :: DFTI_SET_VALUE_INTVAL
     Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
     INTEGER, INTENT(IN) :: Config_Param
     INTEGER, INTENT(IN) :: INTVAL
  END FUNCTION DFTI_SET_VALUE_INTVAL

  FUNCTION DFTI_SET_VALUE_SGLVAL( Desc_Handle, Config_Param, SGLVAL )
     INTEGER :: DFTI_SET_VALUE_SGLVAL
     Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
     INTEGER, INTENT(IN) :: Config_Param
     REAL, INTENT(IN) :: SGLVAL
  END FUNCTION DFTI_SET_VALUE_SGLVAL

  FUNCTION DFTI_SET_VALUE_DBLVAL( Desc_Handle, Config_Param, DBLVAL )
     INTEGER :: DFTI_SET_VALUE_DBLVAL
     Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
     INTEGER, INTENT(IN) :: Config_Param
     REAL (KIND(0D0)), INTENT(IN) :: DBLVAL
  END FUNCTION DFTI_SET_VALUE_DBLVAL

  FUNCTION DFTI_SET_VALUE_INTVEC( Desc_Handle, Config_Param, INTVEC )
     INTEGER :: DFTI_SET_VALUE_INTVEC
     Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
     INTEGER, INTENT(IN) :: Config_Param
     INTEGER, INTENT(IN) :: INTVEC(*)
  END FUNCTION DFTI_SET_VALUE_INTVEC

  FUNCTION DFTI_SET_VALUE_CHARS( Desc_Handle, Config_Param, CHARS )
     INTEGER :: DFTI_SET_VALUE_CHARS
     Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
     INTEGER, INTENT(IN) :: Config_Param
     CHARCTER(*), INTENT(IN) :: CHARS
  END FUNCTION DFTI_SET_VALUE_CHARS
END INTERFACE DftiSetValue
```

```
/* C prototype */
long DftiSetValue( DFTI_DESCRIPTOR_HANDLE,
                   DFTI_CONFIG_PARAM ,
                   ... );
```

GetValue

Gets the configuration value of one particular configuration parameter.

Usage

```
// Fortran
Status = DftiGetValue( Desc_Handle,      &
                       Config_Param,     &
                       Config_Val )
/* C */
status = DftiGetValue( desc_handle,
                       config_param,
                       &config_val );
```

Discussion

This function gets the configuration value of one particular configuration parameter. The configuration parameter is one of the named constants listed in Table 8-1 and Table 8-2, and the configuration value is the corresponding appropriate type, which can be a named constant or a native type.
A zero status value indicates a successful completion. See "Status Checking" for more information on returned status.

Interface and prototype

```
// Fortran interface
INTERFACE DftiGetValue
//Note that the body provided here is for illustration only
//The specification does not mandate what it should be.
//It is possible that implementation will use common external
```

```
//functions below to be shared by Fortran and C implementation of DFTI
 FUNCTION DFTI_GET_VALUE_INTVAL( Desc_Handle, Config_Param, INTVAL )
    INTEGER :: DFTI_GET_VALUE_INTVAL
    Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    INTEGER, INTENT(IN) :: Config_Param
    INTEGER, INTENT(OUT) :: INTVAL
 END FUNCTION DFTI_GET_VALUE_INTVAL

 FUNCTION DFTI_GET_VALUE_SGLVAL( Desc_Handle, Config_Param, SGLVAL )
    INTEGER :: DFTI_GET_VALUE_SGLVAL
    Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    INTEGER, INTENT(IN) :: Config_Param
    REAL, INTENT(OUT) :: SGLVAL
 END FUNCTION DFTI_GET_VALUE_SGLVAL

 FUNCTION DFTI_GET_VALUE_DBLVAL( Desc_Handle, Config_Param, DBLVAL )
    INTEGER :: DFTI_GET_VALUE_DBLVAL
    Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    INTEGER, INTENT(IN) :: Config_Param
    REAL (KIND(0D0)), INTENT(OUT) :: DBLVAL
 END FUNCTION DFTI_GET_VALUE_DBLVAL

 FUNCTION DFTI_GET_VALUE_INTVEC( Desc_Handle, Config_Param, INTVEC )
    INTEGER :: DFTI_GET_VALUE_INTVEC
    Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    INTEGER, INTENT(IN) :: Config_Param
    INTEGER, INTENT(OUT) :: INTVEC(*)
 END FUNCTION DFTI_GET_VALUE_INTVEC

 FUNCTION DFTI_GET_VALUE_INTPNT( Desc_Handle, Config_Param, INTPNT )
    INTEGER :: DFTI_GET_VALUE_INTPNT
    Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    INTEGER, INTENT(IN) :: Config_Param
    INTEGER, DIMENSION(*), POINTER :: INTPNT
 END FUNCTION DFTI_GET_VALUE_INTPNT

 FUNCTION DFTI_GET_VALUE_CHARS( Desc_Handle, Config_Param, CHARS )
    INTEGER :: DFTI_GET_VALUE_CHARS
```

```
    Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
    INTEGER, INTENT(IN) :: Config_Param
    CHARCTER(*), INTENT(OUT):: CHARS
  END FUNCTION DFTI_GET_VALUE_CHARS
END INTERFACE DftiGetValue

/* C prototype */
long DftiGetValue( DFTI_DESCRIPTOR_HANDLE,
                   DFTI_CONFIG_PARAM ,
                    ... );
```

Configuration Settings

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 $k_l = 0, \pm 1, \pm 2, \ldots$, where σ is an arbitrary real-valued scale factor and $\delta = \pm 1$. 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 8-6 for details.

Table 8-6 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. Nevertheless, any one of these four configuration values can be changed, 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.

Sign and scale

The general form of the discrete Fourier transform is given by (7.1), for $k_l = 0, \pm 1, \pm 2, \ldots$, where σ is an arbitrary real-valued scale factor and $\delta = \pm 1$. 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`.

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`.

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.

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

$$z_k = \sum_{j=0}^{n-1} w_j e^{-i2\pi jk/n}, \qquad w_j, z_k \in \mathrm{C}.$$

Assume the stride has default value (unit stride) and DFTI_PLACEMENT has the default in-place setting.

1. DFTI_COMPLEX_COMPLEX storage scheme. 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)` = Re(w_j) , `X(2*j+1)` = Im(w_j) , j = 0,1,...,`n`-1 .

On output,

`X(2*k)` = Re(z_k) , `X(2*k+1)` = Im(z_k) , k = 0,1,...,`n`-1 .

The notations Re(w_j) and 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)` = Re(w_j) , `Y(j)` = Im(w_j) , j = 0,1,...,n-1 .

On output,

`X(k)` = Re(z_k) , `Y(k)` = 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 are recorded in the configuration parameters `DFTI_REAL_STORAGE` and `DFTI_CONJUGATE_EVEN`. Since a forward real domain corresponds to a conjugate even backward domain, we consider them together. We use a one-dimensional real to conjugate even transform as our example. 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

$$z_k = \sum_{j=0}^{n-1} w_j e^{-i2\pi jk/n} , \qquad w_j \in \mathrm{R}, \quad z_k \in \mathrm{C} .$$

There is a symmetry: $z_{n-k} = \overline{z_k}$ whenever $0 \le n\text{-}k < n$. Assume the stride has default value (unit stride).

1. 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 .

2. DFTI_REAL_REAL for real domain, DFTI_COMPLEX_REAL for conjugate even domain. 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,

X(2**k*) = Re(z_k) , X(2**k*+1) = Im(z_k) , *k* = 0,1,...,m .

3. 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* = 0,1,...,m-1 .

4. 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 $\lfloor n/2 \rfloor$.

5. 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,

X(2**k*) = Re(z_k) , X(2**k*+1) = Im(z_k) , *k* = 0,1,...,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,

$Y(k) = \mathrm{Re}(z_k)$, k = 0,1,...,m .

and

$Y(n-k) = \mathrm{Im}(z_k)$, k = 0,1,...,m-1 .

Input and output distances

DFTI allows the computation of multiple number of transforms. Consequently, one 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. 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. Here is the code fragment:

```
COMPLEX :: X(0:31,3), Y(0:64,3)
.................
Status = DftiCreateDescriptor(Desc_Handle, DFTI_SINGLE,
                                  DFTI_COMPLEX, 1, 32)
Status = DftiSetValue(Desc_Handle, DFTI_NUMBER_OF_TRANSFORM, 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)
```

Strides

In addition to supporting transforms of multiple number of datasets, DFTI 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 DFTI 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

x_j = X(1 + L_0 + L_1 * j) = X(5 + L_1 * j) .

This stride vector (2, 4) 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_STRIDE, Stride)
Status = DftiSetValue(Desc_Handle, DFTI_OUTPUT_STRIDE, Stride)
Status = DftiCommitDescriptor(Desc_Handle)
Status = DftiComputeForward(Desc_Handle, X)
```

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

$x_{j_1, j_2, \ldots, j_d}$, $0 \le j_i < J_i$, $1 \le i \le d$

will be stored in the rank-1 array `X` by the mapping

$x_{j_1, j_2, \ldots, j_d}$ = X(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:

$$L_1 = 1, L_2 = J_1, L_3 = J_1 J_2, ..., L_d = \prod_{i=1}^{d-1} J_i$$

Both the input data and output data have a stride associated. The default that is set corresponding to the data to be stored contiguously in memory that is natural to the language.

Finally, consider a contrived example where a 20-by-40 two-dimensional DFT is computed explicitly using one-dimensional transforms.

```
// Fortran
COMPLEX :: X(20,40)
INTEGER :: STRIDE(2)
...
Status = DftiCreatDescriptor( Desc_Handle_Dim1, DFTI_SINGLE,
                                 DFTI_COMPLEX, 1, 20 )
Status = DftiCreatDescriptor( 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_STRIDE, Stride )
```

```
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_OUTPUT_STRIDE, Stride )
Status = DftiCommitDescriptor( Desc_Handle_Dim2 )
Status = DftiComputeForward( Desc_Handle_Dim2, X )

/* C */
float _Complex x[20][40];
long stride[2];
...
status = DftiCreatDescriptor( *desc_handle_dim1, DFTI_SINGLE,
                                DFTI_COMPLEX, 1, 20 );
status = DftiCreatDescriptor( *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_STRIDE, stride );
status = DftiSetValue( desc_handle_dim1, DFTI_OUTPUT_STRIDE, 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 );
```

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. DFTI accommodates 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 DFTI may or may not make use of this setting.

Ordering

It is well known that a number of FFT algorithms apply an explicit permutation stage that is time consuming [4]. Doing away with this step is tantamount to applying DFT to input whose order is scrambled or resulting in scrambling the order of the DFT result. 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. Our API allows the following three options:

1. DFTI_ORDERED: Forward transform data ordered, backward transform data ordered. This is the default.
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 8-7 tabulates the effect on this configuration setting.

Table 8-7 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

$$\text{scram}(i) = i_K + i_{K-1} D_K + i_{K-2} D_K D_{K-1} + ... + i_1 D_K D_{K-1} ... D_2$$

Factoring J into one factor J leads to no scrambling at all, that is, scram(i) = i. Note that the factoring needs not correspond exactly to the number of "butterfly" stages to be carried out. In fact, the computation routine in its initialization stage will decide if indeed a scrambled order in some or all of the dimensions would lead to 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 dimension is higher than one, the next integer value is $K^{(2)}$, etc.

We comment that simple permutation such as mod-p sort [4] is a special case of digit reversal. Hence this option could be useful to high-performance implementation of one-dimensional DFT via a "six-step" or "four-step" framework [4].

We can inquire about the scrambling decided 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

There are FFT algorithms that does not require a scratch space for permutation purposes. We provide 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. Our API offers an option that allows the output be returned in a transposed form if performance gain is expected. 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} , $0 \le j_i < n_i$, 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 option that allows for transposition, the computation may actually return the result into `Y` with stride $L_1 = n_2$ and $L_2 = 1$. These strides are obtainable from an appropriate inquiry function. Note also that in dimension 3 and above, transposition means an arbitrary permutation of the dimension.

Status Checking

All of the functions of the category descriptor manipulation, DFT computation, and descriptor configuration return an integer value denoting the status of the operation. In the status checking category, we provide two

functions. One 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.

Usage

```
// Fortran
Predicate = DftiErrorClass( Status, Error_Class )
/* C */
predicate = DftiErrorClass( status, error_class );
```

Discussion

DFTI provides a set of predefined error class listed in Table 8-8. These are named constants and have the type `INTEGER` in Fortran and `long` in C.

Table 8-8 Predefined Error Class

Named Constants	Comments
`DFTI_NO_ERROR`	No error
`DFTI_INVALID_CONFIGURATION`	Invalid settings of one or more configuration parameters
`DFTI_INCONSISTENT_CONFIGURATION`	Inconsistent configuration or input parameters
`DFTI_BAD_DESCRIPTOR`	Descriptor is unusable for computation
`DFTI_UNIMPLEMENTED`	Unimplemented legitimate settings; implementation dependent
`DFTI_MEMORY_ERROR`	Usually associated with memory allocation

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.

Interface and prototype

```
//Fortran interface
INTERFACE DftiErrorClass
//Note that the body provided here is for illustration only
//The specification does not mandate what it should be.
//It is possible that implementation will use common external
//functions below to be shared by Fortran and C implementation of DFTI
 FUNCTION DFTI_ERROR_CLASS_EXTERNAL( Status, Error_Class )
    LOGICAL DFTI_ERROR_CLASS_EXTERNAL
    INTEGER, INTENT(IN) :: Status, Error_Class
 END FUNCTION DFTI_ERROR_CLASS_EXTERNAL
END INTERFACE DftiErrorClass

/* C prototype */
long DftiErrorClass( long , long );
```

ErrorMessage

Generates an error message.

Usage

```
// Fortran
ERROR_MESSAGE = DftiErrorMessage( Status )
/* C */
error_message = DftiErrorMessage( status );
```

Discussion

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'.

Interface and prototype

```
//Fortran interface
INTERFACE DftiErrorMessage
//Note that the body provided here is for illustration only
//The specification does not mandate what it should be.
//It is possible that implementation will use common external
//functions below to be shared by Fortran and C implementation of DFTI
 FUNCTION DFTI_ERROR_MESSAGE_EXTERNAL( Status, Error_Class )
   CHARACTER(*) DFTI_ERROR_MESSAGE_EXTERNAL( Status )
   INTEGER, INTENT(IN) :: Status
 END FUNCTION DFTI_ERROR_MESSAGE_EXTERNAL
END INTERFACE DftiErrorMessage

/* C prototype */
char *DftiErrorMessage( long );
```

References

[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*, March 2002.

[4] Charles Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM, Philadelphia, 1992

Routine and Function Arguments

A

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 A-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 A-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

$a[m_0], a[m_1], \ldots, a[m_{n-1}]$

and need to be regrouped into an array *y* as

$y[k_0], y[k_1], \ldots, y[k_{n-1}]$,

VML pack/unpack functions can use one of the following indexing methods:

Positive Increment Indexing

$k_j = incy * j,\ m_j = inca * j, \quad j = 0, \ldots, n-1$

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

$k_j = iy[j],\ m_j = ia[j], \quad j = 0, \ldots, n-1,$

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 k_j, m_j are such that:

$my[k_j] \neq 0,\ ma[m_j] \neq 0, \quad j = 0, \ldots, n-1,$

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 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(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*) ≤ *i* ≤ *j*
if *uplo*='L', a_{ij} is stored in *ab*(1+*i*-*j*,*j*) for *j* ≤ *i* ≤ 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 A-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 *

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

B

This appendix presents code examples of using BLAS routines and functions.

Example B-1 Using BLAS Level 1 Function

?dot description

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
```

continued *

Example B-1 Using BLAS Level 1 Function (continued)

```
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 B-2 Using BLAS Level 1 Routine

?copy description

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 B-3 Using BLAS Level 2 Routine

?ger description

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
```

continued *

Example B-3 Using BLAS Level 2 Routine (continued)

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 B-4 Using BLAS Level 3 Routine

?symm description

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
```

continued *

Example B-4 Using BLAS Level 3 Routine (continued)

```
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 B-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 C, "CBLAS Interface to the BLAS".

CBLAS Interface to the BLAS

C

This appendix presents CBLAS, the C interface to the Basic Linear Algebra Subprograms (BLAS).

Similar to BLAS, the CBLAS interface includes three levels of functions:

- Level 1 CBLAS (vector-vector operations)
- Level 2 CBLAS (matrix-vector operations)
- Level 3 CBLAS (matrix-matrix operations).

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.

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?rotg

```
void cblas_srotg(float *a, float *b, float *c, float *s);
void cblas_drotg(double *a, double *b, double *c, double *s);
```

?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);
```

?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);
```

?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);
```

?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);
```

i?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);
```

i?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.

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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.

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

?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);
```

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.
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.
Cholesky factorization	Representation of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A in the form $A = U^H U$ 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 $IA = A$.
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.

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.
MKL	Abbreviation for Math Kernel Library.
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.

positive-definite matrix	A square matrix A such that $Ax \cdot x > 0$ for any non-zero vector x. Here $\cdot$ denotes the dot product.
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.
s	When found as the first letter of routine names, `s` indicates the usage of single-precision real data type.
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 $1.18*10^{-38} < \lvert x \rvert < 3.40*10^{38}$. 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 6 of this book.
z	When found as the first letter of routine names, `z` indicates the usage of double-precision complex data type.

Index

Routines

A

B

C

D

E

F

G

M

Q

R

S

T

U

V

W

X